Refactoring for dynamic languages

Rafael Reia

Instituto Superior Técnico, Universidade de Lisboa
rafael.reia@tecnico.ulisboa.pt

Abstract. Dynamic languages are becoming popular and the need for refactoring tools that support refactoring operations for dynamic languages are increasing. This paper shows the difference between static object oriented refactoring tools and the tools made for dynamic languages and it shows the need of a dynamic language refactoring tool concerned with unexperienced users.

1 Introduction

Over time, software artifacts tends to change, in order to develop gradually, to expand while being used and even during development, new requirements appear, the existing ones change, new bugs are found or some critical new features are added.

Because of these changes the artifact starts to drift apart from its original design in order to incorporate all the modifications. Typically these modifications increases the complexity of the artifact making it less readable and harder to change, consequently making the quality lower and the maintenance costs higher.

In order to improve the quality of software, programmers tend to work on the software structure to make it more readable and therefore easier to understand which leads to a better software quality [1], in other words, programmers try to improve the quality of the software by refactoring it.

Refactoring is a transformation or a set of transformations that are meant to improve the program structure without modifying the behavior. Preserving the behavior is important because if the behavior changes, it transforms the program in a different program.

Refactoring is a tedious and error prone activity, because of that it is preferable to use a tool that provides automated support to the refactoring operations that the programmer intends to do, therefore saving time and reducing the possibility of adding errors to a previous correct program.

There are refactoring tools for the different types of languages paradigms such as static, dynamic, object oriented, functional, imperative. However the most difference refactoring tools are between the dynamic and static languages. The use of the refactoring tools were fully adopted by the object oriented and static programming languages with their IDE support, such as Java [1] with the

[1] https://www.oracle.com/java/index.html

IDEs, Eclipse [2], IntelliJ [3] or NetBeans[4] and C# with Visual Studio [5] but when compared to the dynamic languages there is a lack of refactoring tools and refactoring operations.

The major difference of the refactoring tools made specifically for the dynamic languages when compared with the ones made for static typified languages is the lack of information available, during the refactoring, about the program that is being refactored, such as the type of variables, parameters and returns. This difficulty made the refactoring tools for dynamic languages not evolve like the ones made for static languages. Therefore making the refactoring tools for static languages largely used and considered a common tool unlike for the dynamic languages.

Despite of that, the importance of dynamic languages is growing. Mainly because they are growing very fast among unexperienced programmers, they are used a lot among the scientific community and there are new dynamic languages everyday. The dynamic languages are good for creating prototypes because it is easy and fast to create one. Finally they are easy to adopt and often used as a learning language, such as Scheme, Racket and Python.

The purpose of this paper is to show what work is done, which refactoring tools exist for dynamic languages and to purpose a solution/architecture of a refactoring tool for dynamic languages focused on people that are learning how to program.

2 Related Work

2.1 Use of static refactoring tools

Murphy et al [6] collected some data sets in order to understand how the users refactor. The first data set that was called User data set was collected by Murphy and colleagues [5] in 2005, it has records of 41 volunteer programmers using eclipse 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 04/08 to 01/09 but also include non-Java developers.

The Toolsmiths data set that consists in information about 4 developers who primarily maintain eclipse's refactoring tools. From 12/05 to 08/07 however it is not publicly available and not described in other papers, there is only a similar study by Robbes [9] that uses data from Robbes itself and another developer.

There were some important informations that could get from this paper.

The first one is the most common refactoring operations used by the users with some importance are: Rename, Extract Local Variable, Inline, extract

[2] https://eclipse.org/
[3] https://www.jetbrains.com/idea/
[4] https://netbeans.org/
[5] https://www.visualstudio.com/

Method and Move. The use percentages of this 5 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 was drastically different between toolsmiths and other sets. Toolsmiths had 29% while the User set and Everyone had 62% and 75% respectively.

Using the data sets of users and toolsmiths it was possible to extract 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 where made. In the toolsmith data set only 2 weeks of the year 2006 did not had any refactoring operation and it had in average 30 refactoring operations per week. In 2007 every week had refactoring activities and the average was 47 refactoring operations a week.

Besides refactoring operations being frequent, the refactoring tools are underused. In order to decide whether the refactoring operation was manual or tool assisted they tried to correlate refactoring activities with tool support, if the refactoring activities is correlated with tool support it is classified as being a tool assisted refactoring. 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 this numbers are computed from the toolsmiths data set which is in theory the group who knows and better uses the refactoring tools.

2.2 Overview of static Refactoring tools

The **Table. 1** compares this refactoring tools with each other and the Refactoring definitions can be found on the **Table. 3 in the Apendix A**. This table only lists the refactoring operations that each IDE has by default in order to have a fairer way to compare them with each other. It is easy to see that IntelliJ have 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, the JBuilder have the most used ones as showed above. Only Visual Studio has only 2 out of 5 more used refactoring operations available by default, but there are easily installed plug-ins that cover the more important refactoring operations.

It is easy to see all the effort to provide to the programmers refactoring operations to help them refactoring their projects.

2.3 Griswold

Griswold [4] proved that meaning preserving restructuring can substantively lower the maintenance cost of a system and proved the concept by creating restructuring operations for the Scheme programming language implemented in Common Lisp.

Griswold starts comparing automated restructuring with manual restructuring using an experiment. An initial program and a description of the four goals of

Table 1. Refactoring operations available by default

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

the transformations to be made was presented to 6 subjects. It was also asked to each subject to make sure and demonstrate that the modifications were correct.

Even though that they tested with a really small number of subjects they managed to get several conclusions on how people manually restructure the programs. People use the Copy/Paste and the Cut/Paste paradigm to do the transformations, they copy or cut the code and then paste it in the desired location. The Cut/Paste paradigm is more dangerous because if the user makes any error it is be more difficult to correct the error because it is a destructive operation.

This study also managed to conclude that people make mistakes even with small and simple programs and the cost of correcting mistakes is higher than the time to do the restructuring itself. One last conclusion was that manual restructuring is haphazard, the transformations were done in almost a random way when compared to the computer-assisted process.

It was implemented simple restructuring operations to prove the concept such as: Moving an expression, Renaming a variable, Abstracting an expression, Extracting a function, Scope-wide function replacement, Adding a reference indirection and Adding looping to list references.

In order to be able to implement correctly this operations it was used contours and a PDG (program dependence graph). The main representation of the program is the contours that are an abstraction of the essential semantic properties that the AST (abstract syntax tree) represents in an efficient and complete form, but the PDG does not. The PDG is used to ensure formally that the refactoring is correct. With contours and a way to relate components in the PDG and the AST it is possible to combine then and have a single formalism to reason effectively about flow dependencies and scope structure.

2.4 Refactoring JavaScript

Feldthaus, Asger, et al [2] create a refactoring tool for JavaScript because there are very few refactoring tools for JavaScript. One problem mentioned that might be responsible for that is the additional difficulty that the refactoring tools have to deal with when compared with refactoring tools made for static languages, namely when refactoring JavaScript they do not have information about the bindings in compile time.

The framework uses pointer analysis to help defining a set of general analysis queries. It also uses under-approximations and over-approximations of sets in a safe way and uses preconditions, that are expressed in terms of the analyses queries, to be able to create a correct refactoring operation. If it is not possible for the framework to guarantee behavior preservations, the refactoring operation is prevented. By using this it is possible to be sure when to do the refactoring operation but has the catch of not make every possible refactoring operations because it is an approximation to the set.

To prove the concept it was implemented three refactoring operations, namely the Rename, Encapsulate Property and the Extract Module.

Because it uses approximations it has a certain percentage of refactorings that the framework will unjustified reject that a manual programmer doing that refactoring would be able to do. However, after testing with 50 JavaScript programs, the overall unjustified rejections was 6.2%. The rejections due to imprecise preconditions was 8.2%. Unjustified rejections due to imprecise pointer analysis was 5.9% for the rename and 7.0% for the encapsulate property.

2.5 Refactoring Tool for Samlltalk

This Refactoring Browser [10] is a refactoring tool for smalltalk programs and the goal was to make refactoring knowed and used for everyone, to quoute 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".*

It was considered that the user of this tool was intelligent and that automated refactorings would not suit them because, for example, if it was fully-automated the tool would generate new classes in order to eliminate duplicated code, by creating an abstract class. However that new abstracted class could not make sense in the abstraction of the problem domain. Instead of doing that, the tool points out possible refactoring operations and let the user decide whether or not to do those operations.

This tool uses the reflective features of smalltalk to help doing the analyzes necessary to ensure that the refactorings preserve the behaviour. It is used to check the preconditions of each refactoring before execution. However there are some conditions that are more difficult to determine statically, such as dynamic typing and cardinality relationships between objects, the tool uses another approach, instead of checking the precondition statically, the preconditions are checked dynamically.

This is done using method wrappers to collect run-time information, the Refactoring Browser does the refactoring and then add a wrapper method to the original method and then, as the program runs the wrapper detects the source code that called the original method. One example of this is the rename method, after applying the refactoring and during the program run, when the old method is called, the browser suspends the execution and go to the location of the code that called the old method and changes that to 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.

2.6 Safe Refactoring

Usually each refactoring contains a number of preconditions in order to preserve the behavior of the program, however most refactoring tools do not implement all preconditions because formally establishing all of them is not simple and refactoring tools allow wrong transformations with no warnings to the user.

One way to guarantee the preservation of the program behavior is using tests, but often tests suits do not catch the behavior changes and they might

also be refactored by the tools since they use the program structure that is being transformed.

So Soares, Gustavo, et al. [11] created a tool for eclipse that receives the source code that a refactroring is applied to, generates unit tests to the code being refactored and in the end it reports if the refactoring is safe to apply or not.

It uses a static analysis that identifies methods in common, a method to be considered that needs to have exactly the same modifier, return type, qualified name, parameters types and exceptions thrown in source and target programmers. After identifying those methods it uses Randoop [7], a tool that generates unit tests for classes within a time limit.

Some dynamic refactoring tools such as Smalltalk refactoring browser and similar ones would improve and remove the test suit limitation by being paired with a tool like this one. However that level of static analysis is not that trivial to achieve and that is a big problem.

2.7 Bicycle Repair Man

Bicycle Repair Man is a Refactoring Tool for Python written in Python and it was based on the ideas of Don Roberts PhD thesis. It is a library that can be added to IDEs and editors to provide refactoring capabilities such as emacs, vi and eclipse, and sublime. However, even having a version from sublime this tool did not improve since 2004.

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

The tool has an AST and it does its own parse so it replaces the Python's parser with its own wrapper to be easier to develop the refactoring operations.

2.8 Rope

Rope is a python refactoring tool written in Python, which works like a python library, it is necessary to import it to the your program, and add the project correctly. In order to make it easier Rope makes some assumptions, it assumes that a Python program only has assignments and functions calls. To gather the information for the assignments it is simple, because 'PyName' objects can handle that. However for functions calls it is necessary to have other approaches in order to obtain the necessary information.

It uses a Static Object Analysis which analyses the modules or scope to get information about functions. It only analysis the scopes when they changed and only analyses the modules when asked by the user because this approach is time consuming. The other approach is the Dynamic Object Analysis that starts working when a module is running, then it collects information about parameters passed to and returned from functions in order to get all the information needed. The main problem with this approach is that this it is slow while collecting information however it isn't slow while accessing the information.

It stores the information in a ‘objectdb.ObjectDB‘. If it needs the information, and there is nothing on the ObjectDB the Static object inference starts trying to infer the object that is returned from it.

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

It has simple refactoring operations such as, Rename, Extract method/local variable, Move, inline, Introduce factory, Change method signature, Transform module to package, Encapsulate field, Replace method with method object and Restructuring (like converting "$a.f$(b)" to "$b.g$(a)" where "a: type=mymod.A").

And it also can: Extract similar statements in extract refactorings, fix imports when needed, preview refactorings, undo/redo refactorings, interrupt refactorings, perform cross-project refactorings, handle basic implicit interfaces in rename and change signature.

2.9 Conclusions

With this (**Table. 2.9**) it is possible to confirm that the refactoring tools for dynamic languages are still far away from the capabilities offered by the refactoring tools for static languages and in average have less refactoring operations than the refactoring tools for static languages. One advantage of the refactoring tools for dynamic languages is the interoperability, the majority can be used in different text editors whereas the refactoring tools for static languages are made only for one editor.

	Dynamic?	Refactoring Coverege	Interoperability	Major drawback
Griswold	Yes	Medium		
SmallTalk	Yes	Medium	No	Dependent of Unit tests & maybe dead
Javascript	Yes	Very Low		Few Refactoring Operations
Bicycle	Yes	Medium	Yes	Did not improve since 2004
Rope	Yes	Medium	Yes	
Eclipse	No	High	No	
Visual Studio	No	Medium	No	
IntelliJ	No	High	No	
NetBeans	No	Low	No	
JBuilder	No	Medium	No	

Table 2. Comparassion between Refactoring tools

3 Architecture

There is a lack of refactoring tools for dynamic languages, and none of the existent ones cares about unexperienced users that are starting to program. The idea is to create a refactoring tool that would suit an unexperienced programmer

that is learning how to program. It should be a tool that is simple to use and that the programmer is not afraid to use the refactoring tool. Because of this concerns, the choice of IDE is important, that is why that the idea is to create a refactoring tool for Racket implemented in DrRacket.

DrRacket is an integrated development environment (IDE), that was formerly known as DrScheme. It is a simple IDE that was initially build for Racket programming language and it is aimed at inexperienced programmers. (**Fig. 1**). It was designed as a pedagogic environment [3] and it was used in many intro-

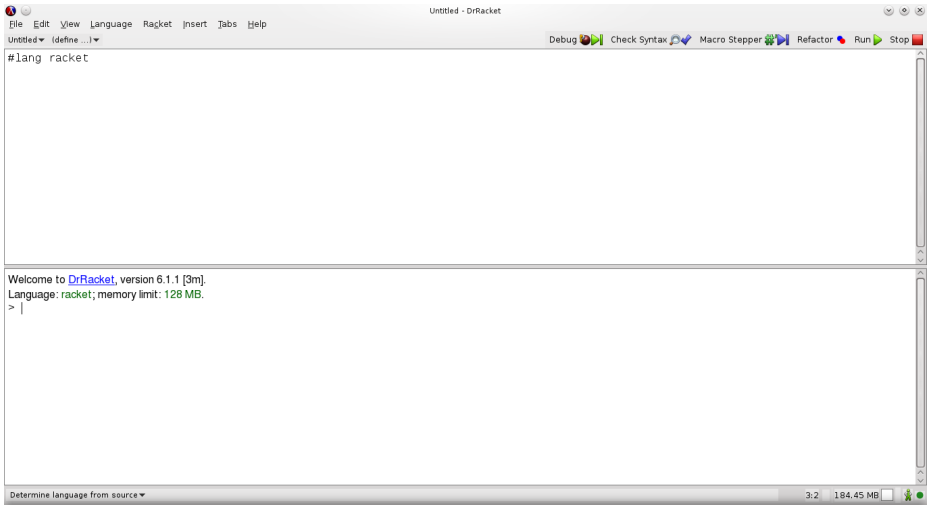

Fig. 1. DrRacket Graphical user interface

ductory programming courses in schools around the world. In addition to that DrRacket supports development and extension of other programming languages [12] and recently it have an implementation of python [8].

Racket[6] programming language is a dialect of lisp and a descendant of Scheme and it supports objects, types and laziness evaluation.

Besides being a simple and a good IDE to learn how to program, DrRacket gives some functionalities that are helpful to the programmer of the refactoring tool. DrRacket knows all the bindings of each variable and represents that binding with arrows. Each arrow knows where is pointing to and where is the beginning of itself. This is rather helpful for the refactoring operations that need semantics, for example for the extract-function it makes it very easy to know what variables will be arguments of the extracted function and what variables are not needed to be passed as arguments.

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

DrRacket also have syntax objects, these objects have all the information regarding the syntactic information and their location. Basically they represent an annotated AST with the location of each object.

4 Conclusions

Refactoring operations are important in order to maintain the quality of the software and because they are error prone and time consuming refactoring tools are needed. The objective of this overview is to show what exist regarding refactoring tools for dynamic languages and the difference between those tools and tools made for static object oriented languages.

Apart from almost every tool made for static object oriented languages have the refactoring operations for 90% of the refactorings done but in the best case scenario only 27% of the refactoring operations were tool assisted.

It also shows how less refactoring operations exist in general for dynamic languages and why it is more difficult to make a refactoring tool for dynamic languages.

In conclusion this paper to show the importance of having a refactoring tool concerned in the unexperienced users. Besides having dynamic tools as the recommended languages to learn how to program, the dynamic languages are growing. By having such tool made for with that concern it might help the users start using the refactoring tools and refactoring their programs alongside with programming.

A Appendix

Table 3. Refactoring operations definitions:

Rename	Renames the selected element and corrects all references.
Move	Moves the selected elements and corrects all references.
Change method signature	Changes parameter names, parameter types, parameter order and updates and all references to the corresponding method.
Extract method	Creates a new method containing the statements or expression currently selected and replaces with a reference to the new method.
Extract local variable	Creates a new variable assigned to the expression currently selected and replaces with a reference to the new variable.
Extract constant	Creates a static final field from the selected expression and,substitutes a field reference, and rewrites other places where the same expression occurs.
Inline	Inline local variables, methods or constants.
Anonymous to nested	Converts an anonymous inner class to a member class.
Move type to new file	Creates a new Java compilation unit for the selected member type updating all references as needed.
Variable to field	Turn a local variable into a field.
Extract superclass	Creates a new abstract class, changes the current class to extend the new class and moves the selected methods and fields to the new class.
Extract interface	Creates a new interface with a set of methods and makes the selected class implement the interface.
Supertype where possible	Replaces occurrences of a type with one of its supertypes after identifying all places where this replacement is possible.
Push down	Moves a set of methods and fields from a class to its subclasses.
Pull up	Moves a field or method to a superclass and in the case of methods declares the method as abstract in the superclass.
Extract class	Replaces a set of fields with new container object. All references to the fields are updated.
Introduce parameter	Replaces an expression with a reference to a new method parameter and updates all callers of the method to pass the expression as the valueof that parameter.
Introduce indirection	Creates a static indirection method delegating to the selected method.
Introduce factory	Creates a new factory method, which calls a selected constructor and returns the created object. Update the references.
Encapsulate field	Replaces all references to a field with getter and setter methods.
Generalize declared type	Allows the user to choose a supertype of the reference's current type.
Type Migration	Change a member type and data flow dependent type entries, across the entire project.
Remove Middleman	Replaces all calls to delegating methods with the equivalent direct calls.
Wrap Return Value	Creates a wrapper class that includes current method return value.
Safe Delete	Finds all the usages of the selected symbol or, simply delete the symbol if no usages found.
Replace Method duplicates	Finds all the places in the current file where the selected method code is fully, repeated and change to corresponding method,calls.
Static to instance method	Converts a static method into an instance method with an initial method call argument being a prototype of newly created instance method call qualifier.
Make Method Static	Converts a non-static method into a static one.
Interface where possible	Used fter using Extract an Interface then search for all places where the interface can be used instead of the original class.
Inheritance to delegation	Delegate 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.

References

1. Fabrice Bourquin and Rudolf K Keller. High-impact refactoring based on architecture violations. In *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, pages 149–158. IEEE, 2007.
2. Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. *ACM SIGPLAN Notices*, 46(10):119–138, 2011.
3. Flanagan C. Flatt M. Krishnamurthi S. & Felleisen M. Findler, R. B. DrScheme: A pedagogic programming environment for Scheme. *Programming Languages: Implementations, Logics, and Programs*, 12(2):369–388, 1997.
4. William G Griswold. Program restructuring as an aid to software maintenance. 1991.
5. Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.
6. Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 38(1):5–18, 2012.
7. Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 75–84. IEEE, 2007.
8. Pedro Palma Ramos and António Menezes Leitão. An implementation of python for racket. In *7 th European Lisp Symposium*, 2014.
9. Romain Robbes. Mining a change-based software repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 15. IEEE Computer Society, 2007.
10. Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Urbana*, 51:61801, 1997.
11. Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *Software, IEEE*, 27(4):52–57, 2010.
12. Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.