Refactoring for dynamic languages

Rafael Reia

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

Abstract. Dynamic languages are becoming popular, specially among unexperienced programmers, and the need for refactoring tools for dynamic languages is increasing. Static language such as Java have several refactoring tools, however, there is a lack of refactoring tools for dynamic languages. This paper present an overview of the refactoring tools for dynamic languages, such as Scheme, JavaScript, Smalltalk, Python and Racket.

Keywords: Refactoring tools, Dynamic languages, Static languages

1 Introduction

Over time, software tends to change, the requirements change or new requirements appear and even after development new bugs are found or some critical new features are added.

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

Refactoring is a transformation or a set of transformations that are meant to improve the program structure and the software quality[1], without modifying its meaning. Preserving the meaning is important because if the meaning 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.

The use of the refactoring tools is fully adopted by the object oriented and static programming languages with their IDE support, such as Java with the IDEs, Eclipse[1], IntelliJ[2] or NetBeans[3] and C# with Visual Studio[4] but when compared to the dynamic languages there is a lack of refactoring tools and refactoring operations.

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

The lack of information available about the program, during the refactoring, is the biggest difference between refactoring tools for static languages and refactoring tools for dynamic languages. This difference is the main difficulty that made the refactoring tools for dynamic languages not evolve like the ones made for static languages. Therefore making the refactoring tools for static languages largely used and considered a common tool in contrast with the refactoring tools for dynamic languages.

Despite 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 which refactoring tools exist for dynamic languages and to propose a refactoring tool for dynamic languages focused on people that are learning how to program.

2 Related Work

The following section firstly presents the use of refactoring tools and an overview of static refactoring tools. Then it presents the refactoring tools for the dynamic languages such as, Scheme, JavaScript, Smalltalk, Python and Racket. Finally it has a conclusion about the related work.

2.1 Use of static refactoring tools

Understanding how users refactor and use the refactoring tools is an important step to better improve the refactoring tools. The information necessary to reason about how the users refactor was gathered by collecting some data sets.[2]

The first data set that called User data set was collected[3] 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[4] 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 five 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 sets is 62% and 75% respectively.

Using the data sets of users and toolsmiths it was possible to confirm that refactoring operations are frequent. In the users data set, 41% of programming sessions contained refactoring activities and the sessions that did not have refactoring activities were the sessions where less edits 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. 2**.

The **table. 1** compares the Visual Studio refactoring operations for C# with Refactoring tools that have refactoring operations for Java because the languages are similar. It also compares with the Eclipse CDT refactoring operations for C++ because in some aspects the languages are similar and the refactoring operations compared are similar to the refactoring operations for Java or C#. The table only lists the refactoring operations that each IDE has by default in order to have a fairer way to compare them with each other. It is easy to see that IntelliJ has almost all the refactoring operations in this table, followed by Eclipse and NetBeans. However, even having significantly less refactoring operations available by default than the other tools, JBuilder has the most used ones as shown above. Only Visual Studio has 2 out of 5 most used refactoring operations available by default, but there are easily installed plug-ins that cover the more important refactoring operations.

Table 1. Refactoring operations available by default

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

Table 2. Refactoring operations definitions:

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

2.3 Scheme

Griswold [5] proved that meaning preserving restructuring can substantively reduce the maintenance cost of a system and proved the concept by creating restructuring operations for the Scheme programming language implemented in Common Lisp.

In order to be able to correctly implement these 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 PDG and AST it is possible to combine them and have a single formalism to reason effectively about flow dependencies and scope structure.

2.4 JavaScript

Feldthaus, Asger, et al [6] created a refactoring tool for JavaScript because there are very few refactoring tools for JavaScript. A problem 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. E.g. when refactoring JavaScript the refactoring tools 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 a refactoring operation is valid but it has the catch of not making every possible refactoring operations because it is an approximation to the set.

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

2.5 Smalltalk

The Refactoring Browser [7] is a refactoring tool for smalltalk programs which goal was to make refactoring known and used by everyone.

In order to ensure behavior preservation the tool checks 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. For that operations the tool uses another approach and instead of checking the precondition statically it checks the preconditions dynamically.

The preconditions checks are done using method wrappers to collect runtime information: the Refactoring Browser does 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 change it for 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.

Safe Refactoring The refactoring browser for smalltalk is dependent on unit tests to have a correct refactoring, but that problem can be solved if there is a tool that create the unit tests for the user.

Soares, Gustavo, et al. [8] created a tool for Eclipse that receives the source code that a refactoring is applied to, generates unit tests to the code being refactored and at 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 [9], a tool that generates unit tests for classes within a time limit.

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

2.6 Python

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

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, Eclipse, and Sublime Text. However, even having a version for sublime this tool did not improve since 2004.

It is an attempt to create the refactoring browser functionality for Python.

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

Rope is a Python refactoring tool written in Python, which works like a Python library. In order to make it easier to create refactoring operations Rope assumes

that a Python program only has assignments and functions calls. The tool can easily get information about the assignments. However for functions calls it is necessary to have other approaches in order to obtain the necessary information.

It uses a Static Object Analysis which analyses the modules or scope to get information about functions. It only analyses the scopes when they are 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 it is slow while collecting information, but not while accessing the information.

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

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

2.7 Racket

Racket[5] programming language is a dialect of lisp and a descendant of Scheme and it supports objects, types and lazy evaluation. For the Racket language the most used IDE is DrRacket. DrRacket is an 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.

Regarding refactoring operations DrRacket only has one, namely the rename. It could be viewed in two ways: the first is they only implemented one refactoring operation and forgot the other ones, or, it could be viewed as they implemented a refactoring operation that is the most used one. In the worst case it represents 29% of the refactoring operations for experienced users and in the best case represents of 62 to 75% for the standard users.

2.8 Conclusions

Dynamic languages, like Racket and Python are used in introductory courses. However even being recognized as a good languages to learn how to program there are very few tools for dynamic languages. Thus the unexperienced programmer only contacts and uses refactoring tools latter on and not when learning the basics.

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.

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

The biggest difficulty for the dynamic refactoring tools is the lack of information available. A dynamic language only knows type information in runtime and that make the refactoring operations more difficult when compared with the refactoring operations for static languages, where the information is always available.

In addition, dynamic languages are very different from each other whereas the static languages are more similar. For example, Java and C# are very similar and even C++ that is not that similar the refactoring operations available for the language are very similar to the refactoring operations available for Java and C#. This difference makes it a bit more difficult for the refactoring tools for dynamic languages when compared to the static one's.

Another difference between refactoring tools is the IDE support. Static refactoring tools have a IDE support for the refactoring operations, thus making it easier when compared to the usual refactoring tools for dynamic languages, that manage all the information. This disadvantage of the dynamic refactoring tools has positive point that they have more interoperability and then easier to use in different text-editors or IDEs.

Even having way less refactoring operations when compared with static refactoring tools, the dynamic refactoring tools at least have the most used refactoring operation, namely the rename.

3 Analyze

It is clear that 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.

Racket is a dynamic language that is known of being used as an introductory programming course in schools around the world. Racket also has an IDE, DrRacket that is a pedagogic environment [10] and it also supports development and extension of other programming languages [11] and recently it has an implementation of python [12]. Besides that, DrRacket only has one refactoring operation, that is the rename. A refactoring tool that would suit an unexperienced programmer who is learning how to program is needed. That will motivate the unexperienced programmers to start using tool assisted refactoring operations and DrRacket seams to be the ideal candidate to have that refactoring tool.

4 Conclusions

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

Although almost every tool made for static languages supports 90% of the refactoring operations, the users only use refactoring tools for 27% of the cases.

It also shows how few 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 shows the importance of having a refactoring tool concerned about the unexperienced users. Besides having dynamic languages as the recommended languages to learn how to program, the dynamic languages are growing. And it might help the users start to use the refactoring tools and refactoring their programs alongside with programming.

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] 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.

[3] Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.

[4] 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.

[5] William G Griswold. Program restructuring as an aid to software maintenance. 1991.

[6] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. *ACM SIGPLAN Notices*, 46(10):119–138, 2011.

[7] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Urbana*, 51:61801, 1997.

[8] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *Software, IEEE*, 27(4):52–57, 2010.

[9] 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.

[10] 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.

[11] 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.

[12] Pedro Palma Ramos and António Menezes Leitão. An implementation of python for racket. In *7 th European Lisp Symposium*, 2014.