# An Automated Rename Refactoring for Go

by

Venkatesh Reddy Burgula

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 10, 2015

Keywords: Rename Refactoring, Golang Rename, Go Refactoring

Copyright 2015 by Venkatesh Reddy Burgula

Approved by

Jeffrey Overbey, Computer Science and Software Engineering (Chair)
Munawar Hafiz, Computer Science and Software Engineering
Sanjeev Baskiyar, Computer Science and Software Engineering

## Abstract

Our work focuses on building an automated Rename refactoring for Go. We show that the tool is effective by testing with a suite of 75 manually written unit test cases and running it on 50 large, open source Go projects from GitHub, which constitute 4 million lines of code.

Our initial implementation used a simple, overly conservative precondition check, where conflicting identifiers in any ancestor or descendent scope flag an error. More robust alternatives have been proposed in the literature. We modified the refactoring tool to use one such alternative, which compares the references of all identifiers before and after refactoring. While this approach is more robust, it is also likely to be more time-consuming. Since refactoring is mostly used by experienced programmers who write clean code, we hypothesize that the more robust solution is unnecessary. To evaluate this hypothesis, we (1) compared the time and space requirements of the original solution and the more robust solution, (2) characterized the conditions under which the original solution produced false positives that were remedied by the more robust solution, and (3) performed an automated analysis of the identifier structure of the 4 million lines of code from GitHub to identify the frequency with which these false positives are likely to occur in practice. Our results show that the precondition-based approach has better performance than the reference-based approach. Our research also found that Go programmers rarely shadow variables, and most shadowing is due to only two variables names: ok and err

**Acknowledgements**

I would like acknowledge all the people who have directly or indirectly helped me through my research. I am immensely grateful to Dr. Jeffrey Overbey, who has been an ideal advisor in every regard. He made me a part of his research team right from my first semester and has been very supportive ever since. I learned a lot from him beyond academics that would not only make me a better engineer but also a good human being. I thank him for all the time that he spent with me to make me learn things and his encouragement throughout my studies at Auburn.

I would like to extend my deepest appreciation for my committee members Dr. Munawar Hafiz and Dr. Sanjeev Baskiyar for inspiring me with their work. They have been very supportive of my work. I owe my position to my parents Yamuna and Chinniah. Finally, I would like to thank my friends and my research team without whom life wouldn't have been easy.

Table of Contents

Abstract ..... ii
Acknowledgements ..... iii
Chapter 1 ..... 1
Introduction ..... 1
1.1 Clean Code ..... 1
1.2 What is Clean Code? ..... 1
1.3 How to Achieve Clean Code? ..... 2
1.4 What is Refactoring? ..... 2
1.4.1 Uses of Refactoring ..... 2
1.4.2 Refactoring Types ..... 2
1.5 Rename Refactoring ..... 3
1.6 Thesis Summary ..... 3
1.6.1 The Problem ..... 3
1.6.2 Thesis Statement ..... 4
1.6.3 Results ..... 4
Chapter 2 ..... 6
Usage of Refactoring Tools ..... 6
2.1 Automated Vs Manual Refactoring ..... 6
What are the most popular automated and manual refactorings? ..... 8
What programmers say about refactoring tools ..... 11
Chapter 3 ..... 13
Study of Identifier Renaming ..... 13
3.1 Standard Approaches to Rename Refactoring ..... 15
3.2 Renaming in static programming languages ..... 16
3.3 Renaming in Dynamic Programming Languages ..... 19
Chapter 4 ..... 21
The Go Doctor ..... 21
4.1 Introduction ..... 21
4.2 Go/AST ..... 21
4.3 Go/ Loader ..... 27
Config ..... 27
Program ..... 27
4.4 Infrastructure for name-search ..... 28

Chapter 5 ........ 32
Go Doctor Rename Precondition based approach ........ 32
5.1 Introduction ........ 32
5.2 Preconditions ........ 33
Input Validity Checks ........ 33
Transformation Checks ........ 35
5.3 Shadowing ........ 36
Chapter-6 ........ 38
Go Doctor Rename References based approach ........ 38
6.1 Introduction ........ 38
6.2 References ........ 38
6.3 Shadowing ........ 39
Chapter 7 ........ 42
Comparison and Results ........ 42
7.1 Time/Space Requirements ........ 42
Time Requirements – Precondition based approach ........ 42
Time Requirements – Reference based approach ........ 44
Space Requirements - Precondition based approach vs References based approach ........ 44
7.2 Empirical Analysis ........ 45
7.3 Conclusion ........ 53
References ........ i

## List of Figures


Figure 1: Relative Proportion of Manual and Automated Refactorings from [5] ... 7
Figure 2: Choice of refactoring by programmers from [5] ... 8
Figure 3: Popularity of Automated Refactorings from [5] ... 9
Figure 4: Popularity of Manual Refactorings from [5] ... 10
Figure 5: Popularity of Refactorings from [5] ... 10
Figure 6: Frequency of Renaming by Developers from [10] ... 14
Figure 7: Rename process by Developers from [10] ... 15
Figure 8: Abstract Syntax Tree for the Go program ... 23
Figure 9: Nodes of AST in figure 8 ... 26
Figure 10: Illustration of Go/loader ... 28
Figure 11: Illustration of Go/loader ... 29
Figure 12: List of keywords in Go from [21] ... 34
Figure 13: List of Predeclared identifiers in Go from [22] ... 35
Figure 14: Map of identifier to declaration in source program ... 41
Figure 15: Map of identifier to declaration in transformed program ... 41
Figure 16: Illustration of Conflict check in Precondition based approach ... 43
Figure 17: Corpus ... 45
Figure 18: Empirical Analysis- Shadowing variables ... 45
Figure 19: Percentage Results of Empirical Analysis - Shadowing variables ... 45

# Chapter 1

# Introduction

## 1.1 Clean Code

Even though levels of abstraction in programming languages increase and number of domain- specific languages continues to grow, code will never disappear. Languages that are close to the requirements and those to help parse and assemble the requirements may be created, but they will never eliminate necessary precision, hence there will always be code. As the mess in code increases the productivity of team work on the code decreases. Efforts to increase the code with the help of additional staff also does not work and may even disrupt the code more. Clean code is cost effective and is matter of professional survival [1].

## 1.2 What is Clean Code?

Here are few regulations that explains clean code. Clean code should be elegant and straight forward. It must have performance close to optimal and should not contain duplicates. Clean code minimizes entities such as classes methods and functions. Clean code should contain meaningful names for all programming entities [1]. However, giving meaningful names to program entities is the most important because 70% of the source code in general consists of identifiers [7].

Here are some of the elementary rules that must be followed by programmers while selecting identifier names. Names of identifiers should reveal its intent without misguiding the programmer. Identifier names should not be misspelled; correcting them may lead to compilation problems. Choose noun phrases for Class names and verbs or verb phrases for method names.

### 1.3 How to Achieve Clean Code?

Automated Refactoring tools are of great aid to achieve clean code, these Refactoring tools save time and effort needed by the programmer to write clean code. Refactoring tools can improve the speed and accuracy with which developers create and maintain software.

### 1.4 What is Refactoring?

A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [2].

Or

To restructure software by applying a series of refactorings without changing its observable behavior [2].

#### 1.4.1 Uses of Refactoring

Refactoring improves the design of software, makes software easier to understand. It also helps to find bugs and write program faster by improves the clarity of program; the time wasted in understanding the program is reduced [1].

#### 1.4.2 Refactoring Types

Refactoring can be applied in two different types (approaches) - root canal refactoring and floss refactoring. If the refactoring is applied in an extended period set a side especially for

refactoring, it is called root canal refactoring. Floss refactoring is done in small steps together with programming tasks, for example, function development [3].

### 1.5 Rename Refactoring

Many studies indicate Rename is the most useful and commonly-used refactoring by programmers. One of such study conducted by Murphy-Hill and Andrew P. Black is published as “Refactoring tools: Fitness for purpose”. The authors presented survey statistics in which 41 developers participated. It shows the statistics of each programmer and number of times a refactoring tool used by the programmer. Survey results show Rename is the most commonly used refactoring tool.

### 1.6 Thesis Summary

Go is a new programming language developed by team at Google and with help of many open source contributors. Types and Interfaces are the core of Go that helps for modular program construction [14]. Its novel type system brings some interesting challenges to program analysis.

Our work focuses on building an automated Rename refactoring for Go. We show that the tool is effective by testing with a suite of 75 manually written unit test cases and running it on 50 large, open source Go projects from GitHub, which constitute 4 million lines of code.

#### 1.6.1 The Problem

Go is statically typed programming language which is easy to learn and has many features that covers the drawbacks of C and C++. It is expected to be widely used in the industry in near future [8]. In this thesis we worked on finding the ideal approach to build the rename refactoring for Go by comparing the traditional precondition based approach with an alternative approach.

Our initial implementation used a simple, overly conservative precondition check, where conflicting identifiers in any ancestor or descendent scope flag an error. We checked for the name

conflicts before applying the transformation by looking for the possible conflicts in all the ancestor and descendant scopes of selected identifier. More robust alternatives have been proposed in the literature. We modified the refactoring tool to use one such alternative, in which collect the references of all identifiers before and after refactoring and check if the Name Binding is preserved. If there is discrepancy in the binding structure of identifiers to its declarations before and after refactoring then only an error is raised .While this approach is more robust, it is also likely to be more time-consuming.

### 1.6.2 Thesis Statement

Our literature survey gives insight that refactoring is mostly used by experienced programmers who write clean code, we hypothesize that the robust solution to build rename refactoring for Go is unnecessary. To evaluate this hypothesis, we (1) compared the time and space requirements of the initial precondition based solution and the more robust references based solution, (2) characterized the conditions under which the precondition based solution produced false positives that were remedied by the references based solution, and (3) performed an automated analysis of the identifier structure of the 4 million lines of code from GitHub to identify the frequency with which these false positives are likely to occur in practice.

### 1.6.3 Results

Our results show that the precondition-based approach has better performance than the reference-based approach. Our research also found that Go programmers rarely shadow variables. Out of 983240 identifiers studied only 1091 (0.1%) of identifiers are shadowed. Although not all shadowing cases create a problem in precondition based approach, shadowing variables increases the potential for occurrences of false positives in precondition based approach and it is not very likely that shadowing occurs in Go so we conclude by saying precondition based method would

be ideal approach. We also observed that most shadowing is due to only two variables names: ok and err.

# Chapter 2

## Usage of Refactoring Tools

Understanding the refactoring practice of developers is important for refactoring tool builders. It helps to improve the current generation of tools or design tools to match the practice, which will help developers to perform their daily task more effectively [4].

Emerson Murphy-Hill and Andrew P. Black [4] conducted a survey on the usage of Refactoring tools and found that Refactoring tools are underutilized by programmers. Most of the tools built in the time frame supported root canal refactoring, but floss refactoring is a popular strategy and a central part of agile methodologies. His Research work suggested that future work on refactoring tools should pay more attention to floss refactoring.

### 2.1 Automated Vs Manual Refactoring

In the paper "A Comparative Study of Manual and Automated refactorings" authors presented an empirical study of refactoring while employing continuous analysis. Continuous Analysis tracks code changes as soon as they happen rather than inferring from Version Control System snapshots. This empirical study answered some interesting research questions given below, which aid a lot to refactoring tool developers [5].

- What is the proportion of manual vs. automated refactorings?
- What are the most popular automated and manual refactorings?

**What is the proportion of manual vs. automated refactorings?**

When investigated deeply to find out the ratio of using automated refactoring tool over applying the refactoring manually, participants of the survey performed around 11% more manual refactorings than using automated refactoring tools.

The usage of refactoring tools also was dependent on the type of refactoring applied. Here are the results of the survey.

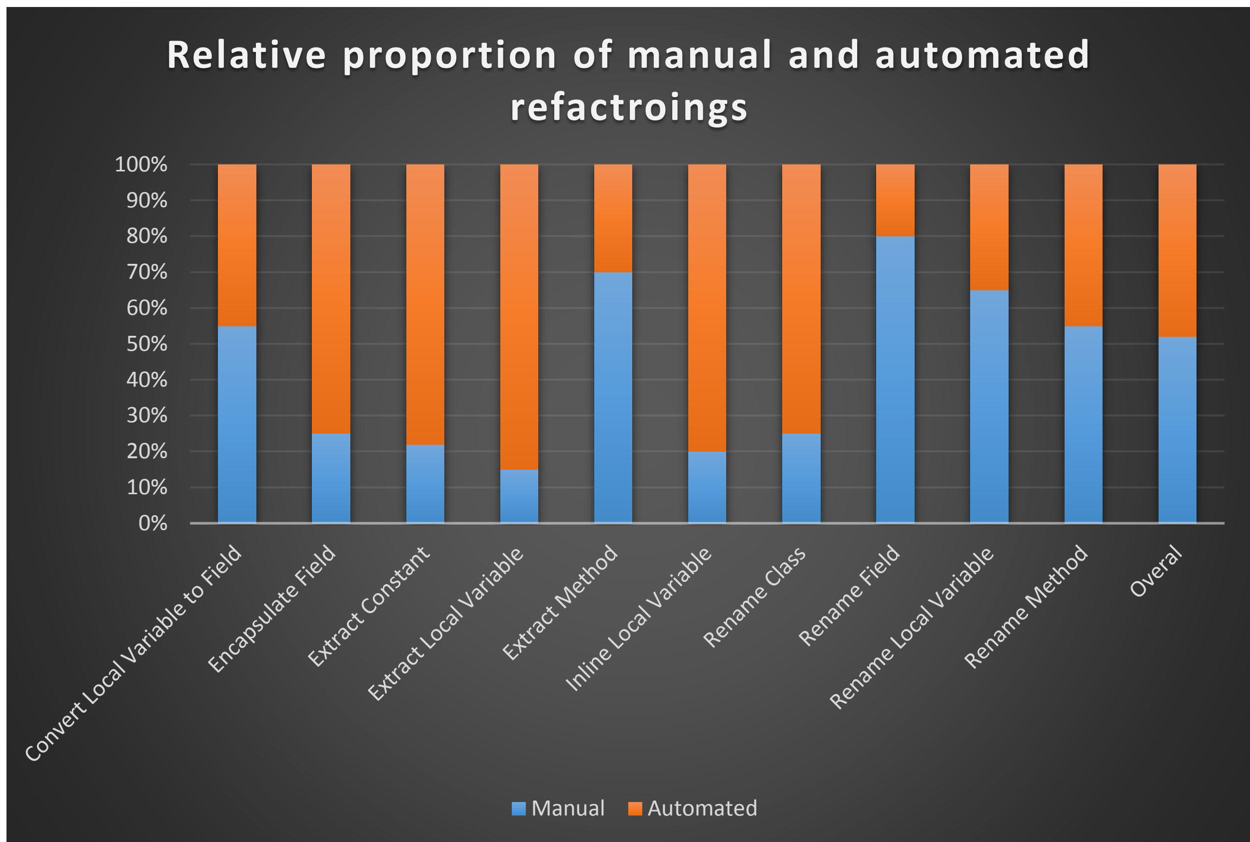



Figure 1: Relative Proportion of Manual and Automated Refactorings from [5]

Half of the refactoring kinds that are investigated, Convert Local Variable to Field, Extract Method, Rename Field, Rename Local Variable, and Rename Method, are primarily performed manually.

The study also considered programmers experience and found that experience of programmers had a great impact on ratio of performed manual and automated refactorings.

The table below shows results of the survey segregated according to the programmers' experience.

| Experience | Manual | Automated | Manual over Automated |
|---|---|---|---|
| <5 years | 292 | 228 | 28% |
| 5-10 years | 1282 | 1459 | -12.1% |
| >10 years | 1237 | 829 | 49.2% |

Figure 2: Choice of refactoring by programmers from [5]

As shown in the table developers with less than 5 years of experience perform 28% more manual than automated refactorings, while the programmers with 5 to10 years of experience perform more automated refactorings than manual. However, developers with more than ten years of experience perform many more manual than automated refactorings (49%). One of the reasons for this behavior could be that more experienced developers learned to perform refactorings well before the appearance of the refactoring tools. Also, experts might think that they are faster without the refactoring tool.

**What are the most popular automated and manual refactorings?**

Popularity of refactorings is shown in three different ways.

- Popularity of the automated refactorings considering only automated refactorings done by the participants.
- Popularity of the manual refactorings considering only manual refactorings performed by the participants

- Popularity of the refactoring as whole considering all the refactorings performed by all participants.

The graphs below show the statistics of popularity of refactorings

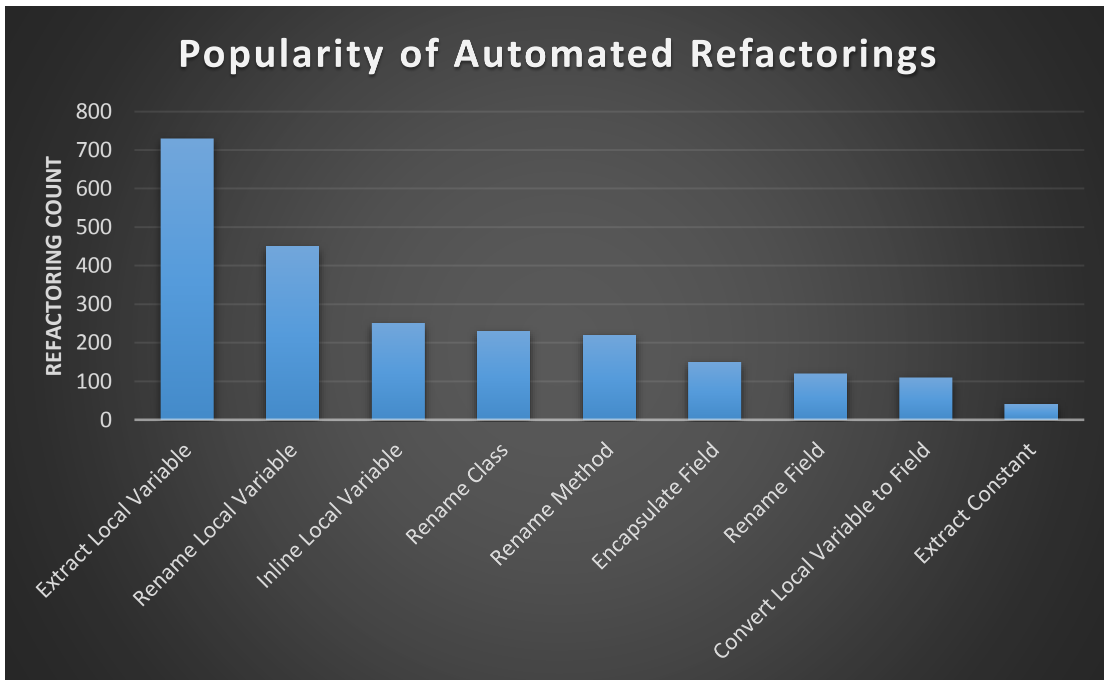



Figure 3: Popularity of Automated Refactorings from [5]

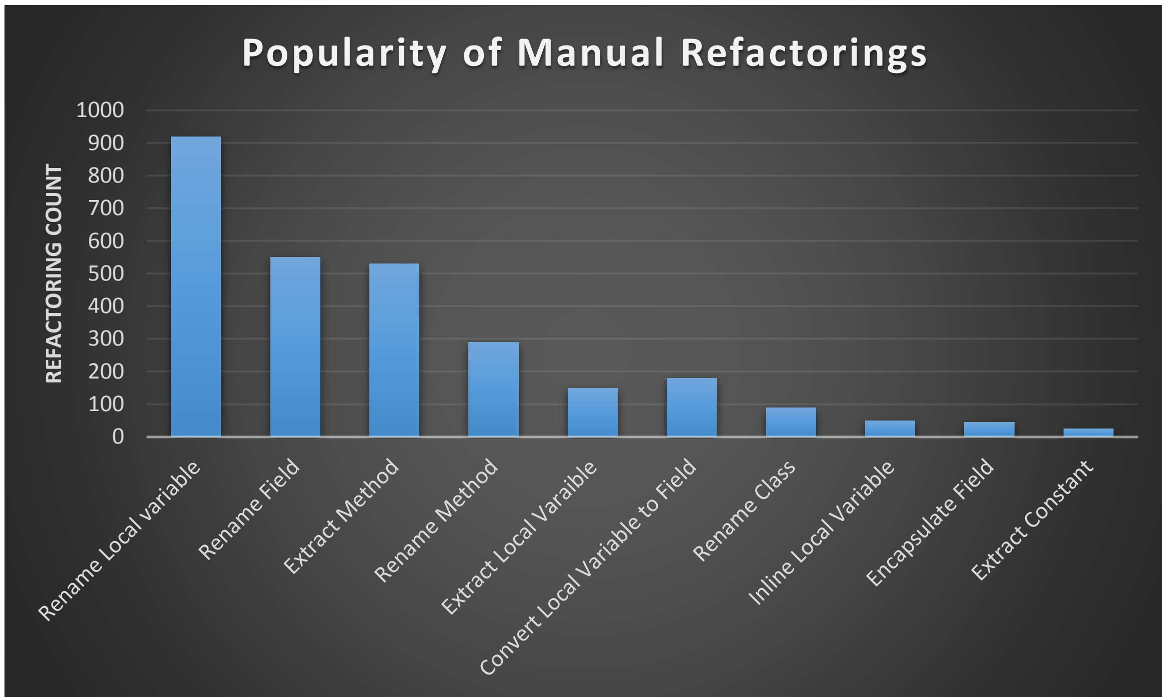



Figure 4: Popularity of Manual Refactorings from [5]

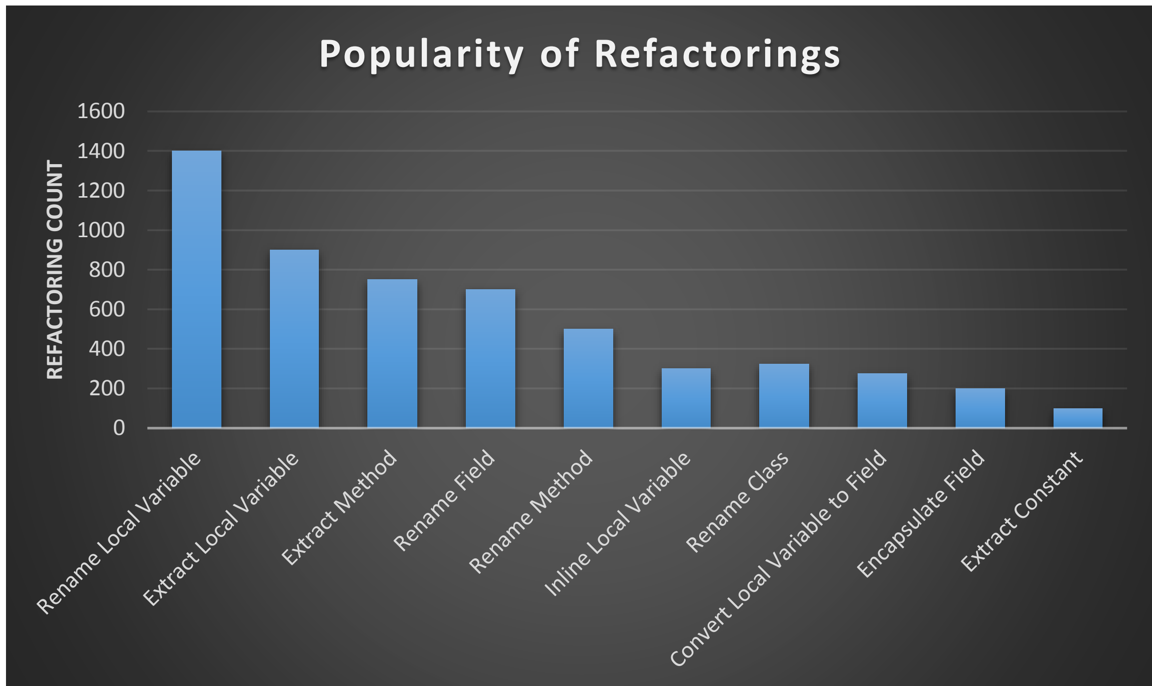



Figure 5: Popularity of Refactorings from [5]

Clearly from Figures 3, 4, and 5 we can see that irrespective of the method used to refactor code, Rename is the most commonly used refactoring by programmers.

**What programmers say about refactoring tools**

The most recent Empirical Investigation of stack overflow by Gustavo Pinto and Fernando Kamei [6] presents different aspects of what programmers say about refactoring tools. This survey is an empirical study of question and answers related to refactoring tools, they analyzed 1400 messages, 324 questions and 1115 answers to those questions from more than 1200 users. Knowledge obtained from the survey is translated into four research questions.

- What are the most desirable features in refactoring tools?
- What are the barriers to adoption of refactoring tools?
- How is interest in refactoring tools related to programmer expertise?
- Does interest in refactoring tools increase over the years?

Qualitative analysis of data gave interesting answers to the questions.

Desired features in refactoring tools include

- Refactoring for dynamic languages
- Refactoring recommendations
- Refactoring for databases
- Multi-language refactoring.

Barriers to adoption of refactoring tools are

- Many refactoring tools are unknown or difficult to learn and use
- Lack of trust

Interest in Refactoring tools is measured in terms of questions and answers posted by programmers with various reputation points. The number of questions posted increases with the increase in reputation of the programmer till the reputation range reaches 1k-10k. On the other

hand, the answers to these questions increase with the increment in reputation, up to the reputation reaches the last group (100K+) as expected. It is notable that members with reputation ranging from 1K to 10K provide the most questions and answers in Stack Overflow.

When the researcher tried to correlate the user's reputation with questions and answers created in Stack Overflow, the questions ratio produced a small correlation with the reputation ratio, indicating that the availability of more expertise increases both the quantity and the quality of questions. Second, the answers ratio produced also a small correlation, indicating that the availability of more expertise increases the number of answers per question.

In an attempt to find if the interest in refactoring tools increase over the years, they noticed that questions and answers had different behavior. The number of questions seem to be quite similar when compared with the firsts months of Stack Overflow, with few variations. The number of answers per questions can vary greatly from quarter to quarter, with some quarters 3 times more activists than others.

Thus from the above observations it is clear that experienced programmers tend to use refactoring tools more than novice programmers.

# Chapter 3

## Study of Identifier Renaming

Approximately 70% of the code of a software system consists of identifiers, hence identifiers play a crucial role in readability and understandability of the code. In general, all programming languages allow programmers to use almost arbitrary sequences of characters with some restrictions as identifier names. Using a random name which follows the constraints of the programming language often leads to meaningless or misleading names. Coding style guides address this problem, but are limited to suggestions like “identifiers should be self-describing.” As the software evolves, more meaningful or appropriate names must be given to identifiers. Rename refactoring helps to change the names of the identifiers by preserving the program behavior [7].

Many programming languages allow globally visible declarations. Therefore renaming requires a global analysis to find which source files need to be changed. To avoid manual inspection of the entire source code, automated tool support for this name-based refactoring provides an invaluable aid in everyday development. An important property of refactoring is that it is behavior preserving. This is particularly important for global refactorings, such as renaming, where it is not reasonable to manually inspect the performed transformations [9].

To analyze the nature of identifier renamings, Venera Arnaoudova and his team conducted an online survey [10] .This survey collected answers to some interesting research questions like “How often do developers rename?”, and “ Is renaming straightforward ? “ .

The picture below shows answers to the first question

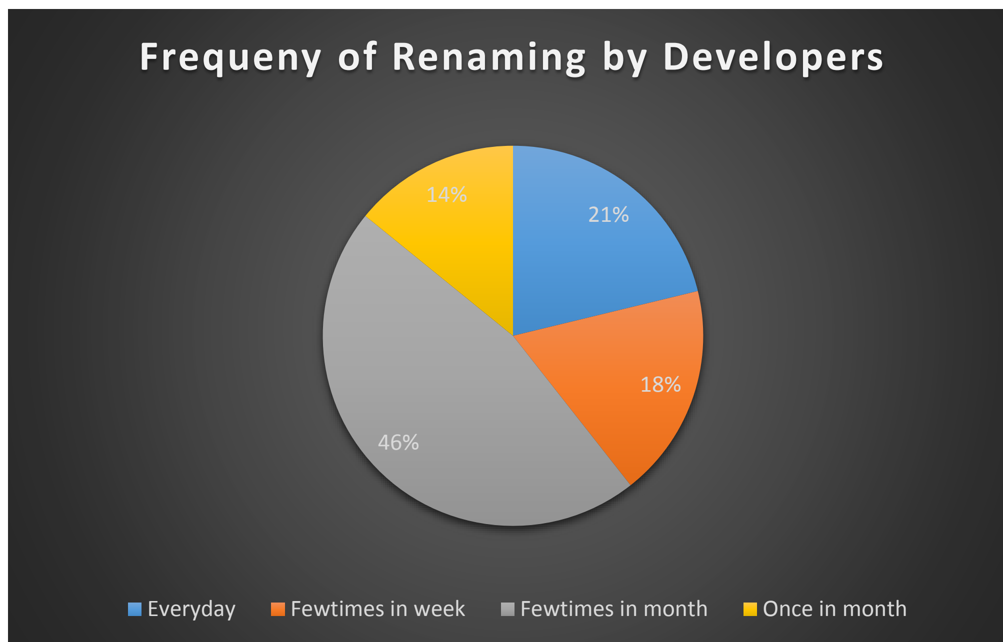



Figure 6: Frequency of Renaming by Developers from [10]

When asked “is reaming straightforward?” only 8% of the people answered yes. Participants’ answers included some interesting facts about using rename refactoring tools. 72% of programmers use automatic tool support for renaming, 20% perform manually and 8% use a mix of both.

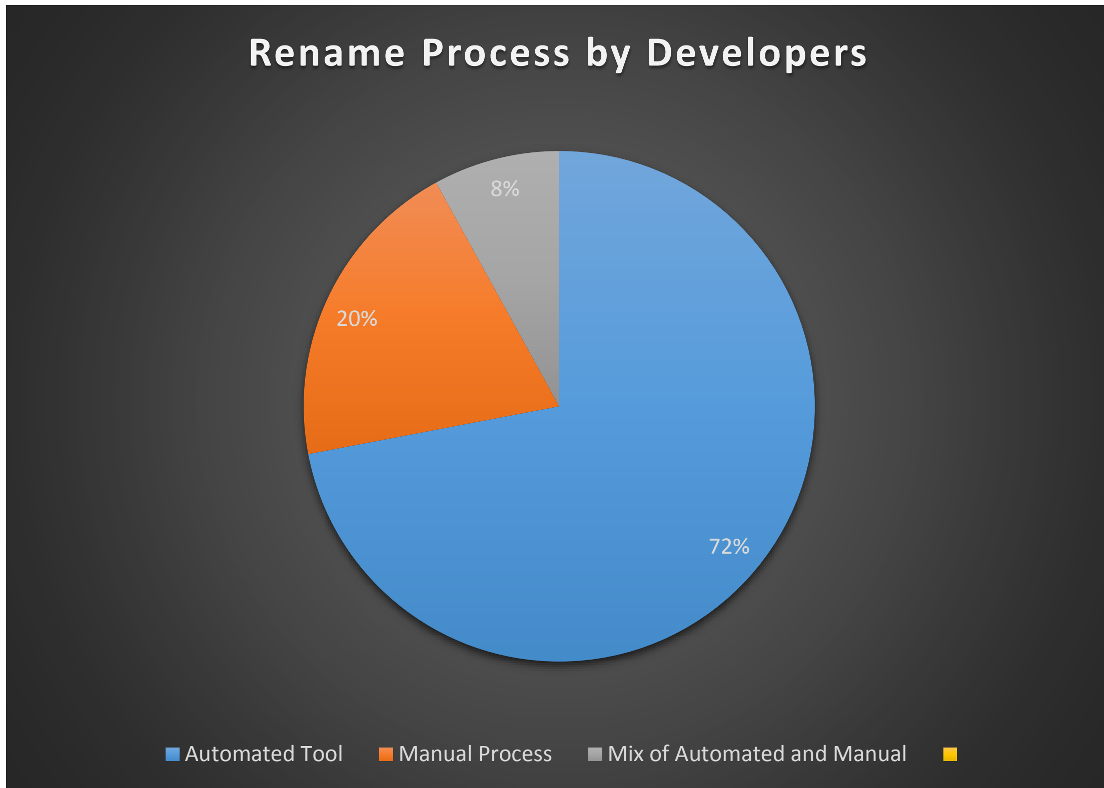



Figure 7: Rename process by Developers from [10]

### 3.1 Standard Approaches to Rename Refactoring

Approaches used to build the rename refactoring for different languages are based on the type of programming language. Generally, programming languages can be classified into two types - static typed and dynamic typed. In statically typed language every variable is bound to both type (at compile time, by means of declaration) and an object, a variable declared to a type is bound to an object of that particular type. Examples of statically typed languages are C, Java, and Go. In dynamically typed languages every variable is bound to the object at run time. Examples of dynamic typed languages are scripting languages like JavaScript, Python, and PHP.

Typically, statically typed languages use static analysis and dynamically typed languages use combination of static analysis and some other means to build the refactoring tools.

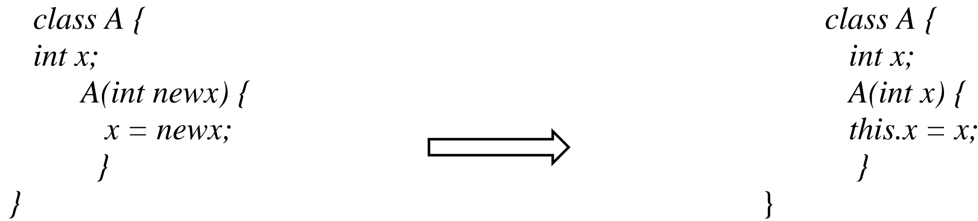
### 3.2 Renaming in static programming languages

The main intention of refactoring is to preserve the behavior and to improve the structure of the existing code. Conventionally, conditions for behavior preservation are implemented as preconditions that are checked before the transformation of the program. Like all refactorings, rename refactorings have been implemented using precondition checks. This approach has several disadvantages that are stated below [11].

- It is very difficult to come up with a correct set of preconditions that guarantees behavior preservation without excluding refactorings that in fact could be carried out.
- Additional preconditions have to be implemented as the language evolves.
- As of now, it is impractical to easily share preconditions among different refactorings or to transfer them to different languages.
- Even mature refactoring frameworks used in current IDEs contain bugs as a result of inadequate preconditions.

In the paper "Sound and Extensible Renaming for Java" Max Schäfer [9] addresses two impediments to a purely precondition-based approach for renaming. Preconditions that are too weak result in unsoundness where programs either do not compile after renaming or variable names in the program refer to different declarations before and after renaming. Preconditions that are too strong would prevent renaming program entities, where some minor qualification would enable the refactoring.

Here is an example that is clearly illustrated by the author, too strong precondition for rename Refactoring used in Eclipse 3.3.1 rejects the refactoring.



From [9]

The above example shows renaming a constructor parameter to the name of field. Since the access to field x would be captured by the renaming, we need to qualify it with this. Eclipse3.3.1 refrain from doing so, and instead rejects the whole refactoring.

When attempted to rename newx to x above, the preconditions were too strong and warned about the name collision and rejected the refactoring. But, refactoring can easily be carried out with only slightly more invasive changes to the user's code.

This is symptomatic of a more general problem with the precondition-based approach to refactoring correctness: preconditions have to be formulated from scratch by reverse-engineering the language specification, and it is very hard to ensure that they cover all problematic cases without forbidding too many useful refactorings [9].

Max Schäfer [9] uses a flexible renaming strategy which is better than using too strong preconditions and systematically avoids too weak preconditions. The strategy uses a correctness criterion that Rename refactorings should preserve the invariant that only names are affected by the refactoring, and that each name refers to the same declared entity before and after the transformation. This correctness criterion is slightly weaker than behavior preservation, which is impossible to achieve in Java under presence of dynamic class loading and reflection [9].

To implement the correctness criterion a new technique for creating symbolic names that are guaranteed to bind to a desired entity in a particular context by inverting lookup functions. By

applying this operation for each bound name that could possibly be affected by a renaming operation, it is guaranteed that the correctness invariant described above is preserved. This also allows to re-qualify names that would otherwise refer to different declarations before and after the refactoring. If no symbolic name can be computed for a desired context, the rename operation aborts and reverts all changes [9].

Name bindings associate identifiers with program entities such as variables, methods and types. Name bindings form a semantic concern that should be preserved by refactorings. Intuitively, all name accesses in a program should bind to the same declarations before and after the transformation [11].

Shadowing occurs when the same identifier is used for different entities in nested lexical scopes. Shadowing poses a challenge for binding preservation in refactorings [11].

Maartje de Jonge and Eelco Visser [11] proposed a language generic (statically typed languages) solution for preserving static name bindings. To make the binding structure explicit, the proposed solution uses technique of explicit renaming implemented in Stratego by means of term annotations. The result of the name analysis is an abstract syntax tree in which all identifiers are annotated with a globally unique reference name. The generated AST is used in a rename refactoring to determine which identifiers must be renamed. The name annotations are used to implement name binding preservation as a post condition on the transformed tree. The preservation condition compares the existing annotations, which represent the original bindings, with the new Annotations, obtained by reanalyzing the transformed tree. Preservation is realized if and only if the existing name annotations are equal to the new name annotations, modulo renaming.

To check the solution's application in the real world they developed a language generic rename refactoring. The tool they built is applied for these three programming languages Stratego, subset of Java language, and Mobl [11].

### 3.3 Renaming in Dynamic Programming Languages

The technique used in developing automated refactoring tools critically depend on static information about class hierarchies, packages, type of fields and methods they cannot be easily adapted to dynamically typed languages [11].

In JavaScript the object properties change at run time, and classes and packages are best mimicked by meta-programming in libraries which are difficult to reason about statically. The Refactoring techniques that exist for dynamically typed languages, in particular JavaScript remain primitive or impractical [11].

Asger Feldthaus and Anders Møller proposed a semi-automatic refactoring for JavaScript [12], with focus on renaming. Unlike traditional refactoring algorithms, semi-automatic refactoring works by a combination of static analysis and interaction with the programmer.

Semi-Automatic technique for renaming object properties is a middle ground between fully automated refactoring and tedious one-by-one search-and-replace. If the programmer decides that some occurrence of an object property x in the program needs to be renamed to y, a lightweight static analysis is used to divide all occurrences of x into groups. Two occurrences of x are placed in the same group if the analysis infers that they are related in the sense that they must either both be renamed or neither be renamed. The refactoring tool then asks the programmer whether each group should be renamed entirely or left unchanged. This way, the programmer gets one question for each group, and each question only mentions one occurrence of x. Compared to traditional search-and-replace, this approach requires less effort by the programmer since the tool avoids

asking questions where the answer can be inferred from previous answers. Compared to the fully automated approach it circumvents the computationally expensive pointer analysis and the whole-program assumption [12].

# Chapter 4

# The Go Doctor

This chapter introduces Go programming language and Go Doctor a refactoring engine for Go. It explains about Abstract Syntax trees and how Go/Loader works. It illustrates the Infrastructure used for name search in our rename refactoring tool.

## 4.1 Introduction

Go is an open source programming language introduced by Google in 2007 [13]. Go is expressive, concise, clean, and efficient [14]. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction [14]. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language [14].

Go Doctor is an Automated Refactoring tool for Google's Go programming language. It performs various refactorings like Rename, Toggle var ⇔:=, Extract Function, Add Godoc, Extract Local Variable. It provides infrastructure for building more refactorings .It is written in Go and can be Integrated into other IDE's or text editors.

## 4.2 Go/AST

Abstract syntax tree is a data structure used to represent the structure of program code [16]. It is in general the result of the syntax analysis phase of compiler [16]. Each node of the tree denotes a construct occurring in the source code [16]. The syntax is "abstract" in not representing every detail appearing in the real syntax [16]. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by

means of a single node with three branches. Package ast declares the types used to represent syntax trees for Go packages [15]. For Example type ident in go/ast package represents an identifier in Go language. [15]

Consider a simple Go Hello World program

```
package main
import "fmt"
func main() {
var localvar string ="Hello World"
fmt.Println(localvar)
}
```

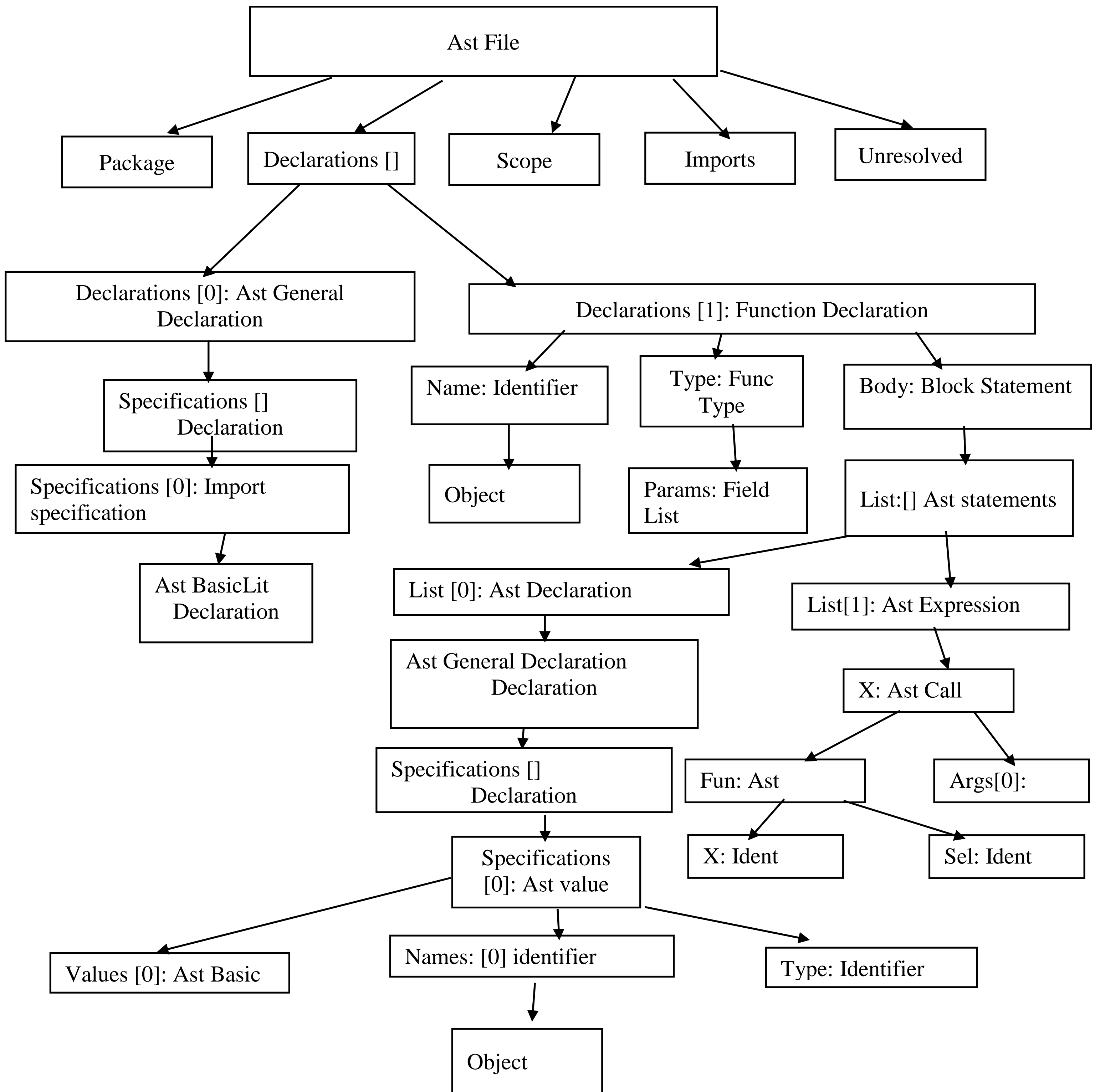



Figure 8: Abstract Syntax Tree for the Go program

| Node # | Node Type | Node |
|---|---|---|
| 1 | *ast.File | package main<br>import "fmt"<br>func main() {<br>var localvar string ="Hello World"<br>fmt.Println(localvar)<br>} |
| 2 | *ast.Ident | Main |
| 3 | *ast.GenDecl | import “fmt” |
| 4 | *ast.ImportSpec | “fmt” |
| 5 | *ast.BasicLit | “fmt” |
| 6 | *ast.FuncDecl | func main() {<br>var localvar string ="Hello World"<br>fmt.Println(localvar)<br>} |

| 7 | *ast.Ident | main |
|---|---|---|
| 8 | *ast.FuncType | func |
| 9 | *ast.FieldList | () |
| 10 | *ast.BlockStmt | {<br>var localvar string ="Hello World"<br>fmt.Println(localvar)<br>} |
| 11 | *ast.DeclStmt | var localvar string ="Hello World" |
| 12 | *ast.GenDecl | var localvar string ="Hello World" |
| 13 | *ast.ValueSpec | localvar |
| 14 | *ast.Ident | localvar |
| 15 | *ast.Ident | string |

| 16 | *ast.BasicLit | "Hello World" |
| --- | --- | --- |
| 17 | *ast.ExprStmt | fmt.Println(localvar) |
| 18 | *ast.CallExpr | fmt.Println(localvar) |
| 19 | *ast.SelectorExpr | fmt.Println |
| 20 | *ast.Ident | fmt |
| 21 | *ast.Ident | Println |
| 22 | *ast.Ident | localvar |

Figure 9: Nodes of AST in figure 8

### 4.3 Go/ Loader

Though the Parser package is the standard package to get AST information of any Go source it does not handle dependencies. For a given program to apply transformation we need to to get AST information of the whole program not just the given file. To load all the dependencies of given file and get AST information of all its dependencies we used Go/Loader [17]. Package loader loads, parses and type-checks packages of Go code and their transitive closure, and gets information about both the ASTs and the derived facts. This package defines two primary types Config and Program.

**Config**

It provides several convenient functions to load, type check and parse the Go program and its dependencies. It has functions that helps to load the dependencies of the program by taking input in various forms like filenames, AST files or package names. It also provides an option to load or not load the program and its dependencies when there are compiler errors present in the given program. After setting the configuration by giving the input program finally a call to load is made to actually load and type check the program [17].

**Program**

It is the result of successfully loading the packages specified by the configuration [17]. In the case of Rename Refactoring Program includes the program in which the identifier is selected and all its dependent files.

The figure below gives a picture of how Go/Loader works

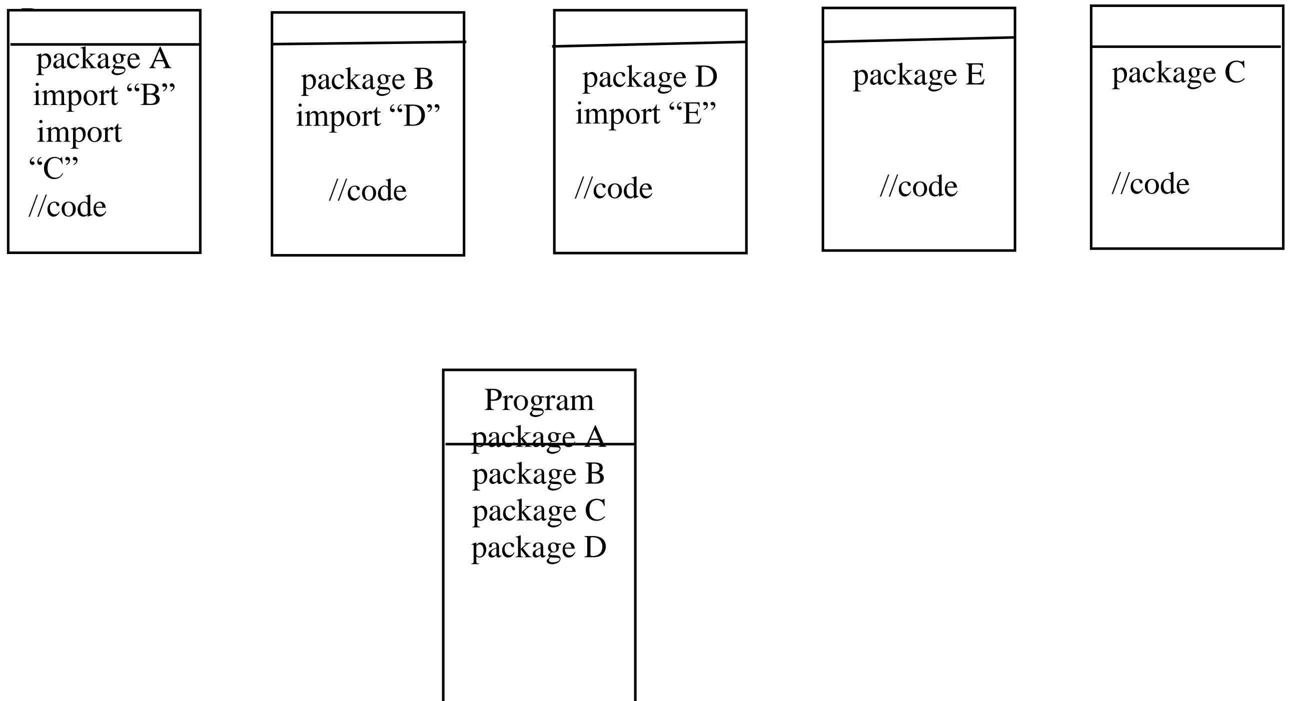



Figure 10: Illustration of Go/loader

### 4.4 Infrastructure for name-search

The names package provides the utility functions to search all the occurrences of the given identifier and to find the conflicts of new name with already existing names. It also has function that helps to find the occurrences of given identifier name in comments.

To find all the identifiers which refers the given identifier, we get object of the selected identifier and check if any other identifier has the same object as the selected one, collect all such occurrences and rename them. Most identifiers are mapped to objects however there are some cases where identifiers do not have objects for example TypeSwitchVar and special cases like

methods where not only identifiers with matching objects but other identifiers or method names should be renamed.

In methods, there may be indirect relationships such as the following shown in Figure 11

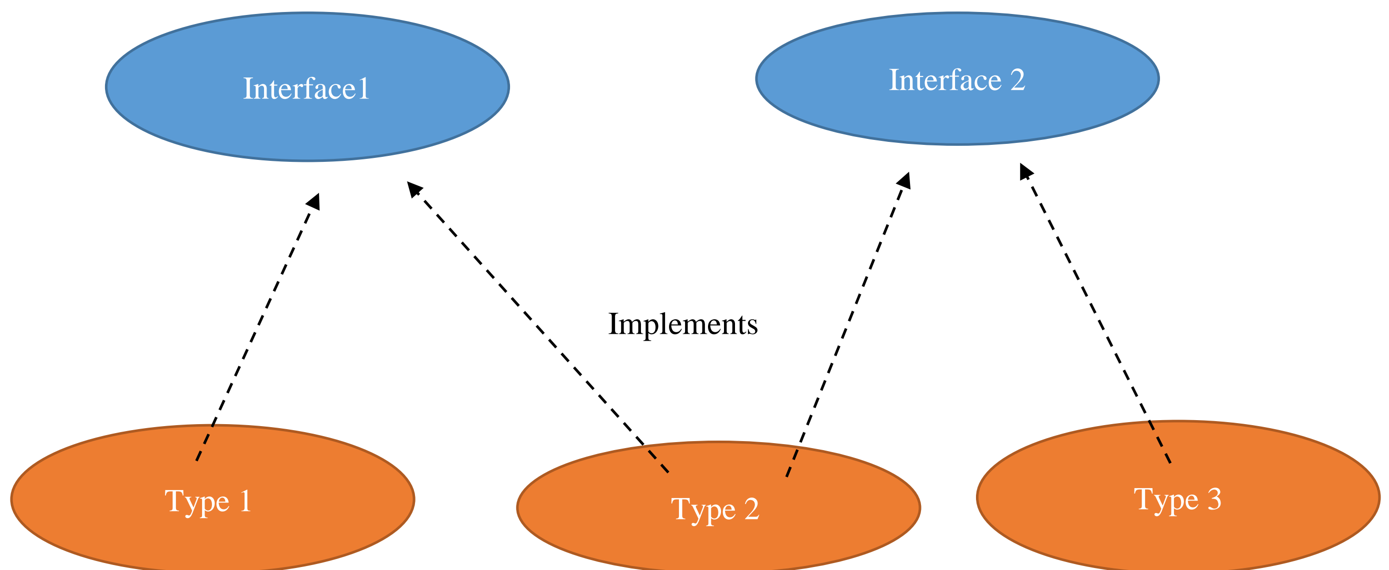



Figure 11: Illustration of Go/loader

Where renaming a method in Type1 could force a method with the same signature to be renamed in Interface1, Interface2, Type2, and Type3.

If the given identifier is a method of Type1 then we search across reflexive transitive closure of interfaces because there may be methods with that need to change to ensure all types continue to implement same interfaces after the refactoring.

Names package also has function to find occurrences of TypeSwitch variables whose objects are nil by definition .It makes use of identifier type information to collect all the identifiers that should be renamed.

FindInComment is utility function that searches the comments of the given packages' source files for occurrences of the given name (as a word, not a sub word) and returns their source locations.

The example below illustrates how the rename refactoring works in case of TypeSwitch.

**source program before applying the rename refactoring.**

```
package main
import "fmt"
// Test for renaming the type switch variable
func main() {
var t interface{}
t = bool(true);
switch y := t.(type) {  // <<<<< rename,11,8,11,8,renamed,pass
default:
   fmt.Printf("unexpected type %T", y)
case bool:
   fmt.Printf("boolean %t\n", y)
case int:
   fmt.Printf("integer %d\n", y)
case *bool:
   fmt.Printf("pointer to boolean %t\n", *y)

case *int:
   fmt.Printf("pointer to integer %d\n", *y)
}
```

```
}
```

**Transformed program after applying the rename refactoring.**

```
package main
import "fmt"
// Test for renaming the type switch variable
func main() {
var t interface{}
t = bool(true);
switch renamed := t.(type) {  // <<<<< rename,11,8,11,8,renamed,pass
default:
   fmt.Printf("unexpected type %T", renamed)
case bool:
   fmt.Printf("boolean %t\n", renamed)
case int:
   fmt.Printf("integer %d\n", renamed)
case *bool:
   fmt.Printf("pointer to boolean %t\n", *renamed)
case *int:
   fmt.Printf("pointer to integer %d\n", *renamed)
}
 }
```

# Chapter 5

## Go Doctor Rename Precondition based approach

Automated refactorings have two parts: the transformation—the change made to the user's source code—and a set of preconditions which ensure that the transformation will produce a program that compiles and executes with the same behavior as the original program [18].

In Section 4.4 we already discussed how to search for the identifiers and apply the transformation. This chapter illustrates how we Implemented the rename refactoring using pre condition checks for all program entities in Go (except packages) and what are the corner cases that could not be addressed using the pre-condition checks.

### 5.1 Introduction

Rename is a refactoring that changes the names of variables, functions, methods, types, interfaces, and packages in Go programs. It attempts to prevent name changes that will introduce syntactic or semantic errors into the program.

Rename refactoring uses the API Refactoring Base which extracts all the information that is primarily needed to perform the refactorings. Refactoring Base makes use of Go/Loader mentioned in the previous chapter to extract the required information from user's selection. Here are some of the key elements of Refactoring Base that will be used in Rename refactoring.

- The program to be refactored, including all dependent files
- The AST of the file containing the user's selection
- The filename containing the user's selection
- The deepest ast.Node enclosing the user's selection
- The package Information containing the selectedNode

Rename refactoring needs an additional parameter apart from selecting the text or name of identifier to rename, this additional parameter is the new name that user wants to give to the selected identifier.

### 5.2 Preconditions

Based on the ideas of Jeffrey L. Overbey and Ralph E. Johnson that were presented in the paper Differential Precondition Checking: A Lightweight, Reusable Analysis for Refactoring Tools, The preconditions we implemented can be logically divided in to two types [18].

- Input Validity Checks

  Checks if all input from the user is legal because it is not possible to apply the transformation to the given program with invalid inputs

- Transformation Checks

  Checks if the transformation is performed then resulting program is compilable and executable and it will exhibit the same runtime behavior as the untransformed program. Some preconditions we used here are unnecessarily conservative and rejects some easily possible refactorings.

**Input Validity Checks**

Given below are the set of preconditions that are implemented in our refactoring tool to validate the users input.

- The deepest ast node enclosing the users selection represented as selectednode should not be nil.
- The new name passed as parameter to the rename refactoring should not be empty.
- The new name should be a valid Go identifier.

- The new name should not be a reserved word in Go. That is it must not be any of the following keywords.

| Break | default | func | interface | select |
|---|---|---|---|---|
| Case | defer | Go | map | struct |
| Chan | else | goto | package | switch |
| const | fallthrough | If | range | type |
| continue | for | import | return | var |

Figure 12: List of keywords in Go from [21]

- main function inside main package is not a valid selection, as renaming it would eliminate program entry point.
- Predeclared identifiers cannot be renamed,  Given below are the set of predeclared identifiers in Go

| Bool | byte | complex64 | Complex128 | error | float32 | float64 |
|---|---|---|---|---|---|---|
| Int | int8 | int16 | int32 | int64 | rune | string |
| uint | uint8 | uint16 | uint32 | uint64 | uintptr | false |
| true | iota | nil | append | close | complex | copy |

| deleted | imag | len | make | new | panic | print |
|---|---|---|---|---|---|---|
| println | real | recover | cap | | | |

Figure 13: List of Predeclared identifiers in Go from [22]

- In Go if an Identifier's first character is a Unicode upper case letter then is it can be accessed outside the package from another package (exported). If selected identifier is exported then new name should be exported.
- The selected node should be an identifier to rename it, else it cannot be renamed.
- The Go binary distributions locations is called GOROOT. If the selected identifier is in GOROOT it cannot be renamed.

**Transformation Checks**

We performed the transformation check with the help of preconditions that check if there already exists an identifier with the given new name such that the selected identifier cannot be renamed to given new name. It returns one such conflicting declaration, if possible, and nil if there are none.

The preconditions check for conflict in the current scope (scope of the selected identifier) or any of its child scope. They also check for the possible conflicts from parent scope of the selected identifier.

If the selected identifier is a method then the precondition tries to find conflicting method with same receiver type.

### 5.3 Shadowing

The transformation precondition checks that are performed to avoid conflicts with existing identifiers though seem to be provide safe transformations ,they are overly conservative and the tool rejects some rename refactorings which can be directly carried without creating conflicts.

Here is an example of one such case, which involves shadowing of global variable in the local scope.

**Source program to refactor**

```
package main
import "fmt"
var largescope = ":-("
func main() {
	largescope = ":-)"

	var hello string = "Hello"
	var world string = "world"
	hello = hello + ", " + world
	hello += "!"
	fmt.Println(hello)
}
```

In the above program if we manually rename hello to largescope, then transformed program would be as shown below.

**Transformed program after renaming manually**

```
package main
import "fmt"
var largescope = ":-("
func main() {
	largescope = ":-)"

	var largescope string = "Hello"
	var world string = "world"
	largescope = largescope + ", " + world
	largescope += "!"
	fmt.Println(largescope)
}
```

The transformed program behavior is compilable, executable and behavior is preserved. However this transformation is not allowed in our rename refactoring tool built with precondition based approach since, the new name (largescope) conflicts with existing variable declaration (largescope) in the parent scope.

To make the refactoring tool for such shadowing cases we modified it to use a new algorithm, which is sound and works on all such cases.

# Chapter-6

## Go Doctor Rename References based approach

This chapter illustrates how we Implemented the rename refactoring using references based approach for all program entities in Go (except packages) and show that it is sound and works for corner cases which could not be addressed in the former method or approach.

### 6.1 Introduction

The implementation of rename refactoring in the reference based approach uses the same set of input validity checks used in the former approach; Given any approach the input should be valid before applying the transformation.

The difference in reference based approach compared to precondition based approach (assuming the given input is valid) - The precondition based approach performs a set of transformation checks, if they pass only then transformation is applied but in the reference based approach we go ahead and apply the transformation then analyze the program after it has been transformed, compare the program to original program to determine whether or not the transformation preserved behavior.

### 6.2 References

As discussed in the introduction, we analyze the program after the transformation and check if the behavior is preserved .The rename refactoring preserves name binding relationship. : It ensures that every identifier refers to the “same” declaration before and after transformation. Our implementation includes the following steps.

- Perform all input validity checks used in precondition based approach

- Collect the references or declarations of all identifiers in the source program
- Search for the occurrences of selected identifier and apply the transformation as discussed in section 4.4
- Collect references or declarations of all identifiers in the transformed program
- Compare the references of identifiers before and after the transformation, if any of the identifier binds to different declarations in the source and transformed programs ,then raise an error

**6.3 Shadowing**

In this section we show how our reference based rename refactoring successfully works on the corner case discussed in section 5.3

Consider the same example program that is shown in section 5.3

**Source program to refactor**

```
package main
import "fmt"
var largescope = ":-("
func main() {
	largescope = ":-)"
	var hello string = "Hello"
	var world string = "world"
	hello = hello + ", " + world
	hello += "!"
	fmt.Println(hello)
}
```

Following the implementation steps mention in section 6.2, for the program

- Input is valid ,all preconditions in the input validity checks are met
- Collect references of all identifiers in the program

**Transformed program after renaming**

```
package main
import "fmt"
var largescope = ":-("
func main() {
        largescope = ":-)"

        var largescope string = "Hello"        // <<<<< rename,11,6,11,6,largescope
        var world string = "world"
        largescope = largescope + ", " + world
        largescope += "!"
        fmt.Println(largescope)
}
```

The table below shows mapping of each identifier to its declaration that is collected

| Identifier name | Declaration it is bound to |
|---|---|
| largescope | var main.largescope string |
| main | func main.main() |
| largescope | var main.largescope string |
| hello | var hello string |
| world | var world string |
| hello | var hello string |
| hello | var hello string |
| world | var world string |
| hello | var hello string |
| hello | var hello string |

Figure 14: Map of identifier to declaration in source program

Next step is find all the occurrences of selected identifier ‘hello’) and rename them to ‘largescope ‘, which is applying the transformation.

Then we collect the references of all the identifiers in the transformed program

The table below shows mapping of each identifier to its declaration that is collected

| **Identifier name** | **Declaration it is bound to** |
|---|---|
| largescope | var main.largescope string |
| main | func main.main() |
| largescope | var main.largescope string |
| largescope | var largescope string |
| world | var world string |
| largescope | var largescope string |
| largescope | var largescope string |
| largescope | var largescope string |
| largescope | var largescope string |
| largescope | var largescope string |

Figure 15: Map of identifier to declaration in transformed program

We compare the binding structure before and after the transformation and see that it is preserved. Hence this transformation is behavior preserving and can be applied

# Chapter 7

# Comparison and Results

In this chapter we compare the precondition based approach with reference based approach in terms of time and space requirements. We discuss our empirical study to find the frequency of occurrence of shadowing variables in Go to understand which approach would be better to build the commercial Rename refactoring.

## 7.1 Time/Space Requirements

As mentioned in earlier chapters, the only difference between two approaches is the part where we do transformation checks. We compare the cost involved in doing the precondition checks that facilitates safe transformation in precondition based approach with the cost of analyzing the program after the transformation in reference based approach.

**Time Requirements – Precondition based approach**

In precondition based approach we try to find the conflicts by checking if there already exists an identifier with the same name as new name given by user in the current scope or any of the child scopes or parent scopes of selected identifier, to do this check we make use of information from symbol table.

A symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location [19].

Algorithm

1. For the selected identifier we first get the scope of it, which is the current scope. Consider current scope as the root node of the tree on which we perform depth first search.
2. look in scope to see if there is an identifier with new name ,if yes then raise an error
3. if the current scope ( except when current scope is root node) has no children then go back to its parent scope ,and start with another child scope which is sibling of the current scope else get all the child scopes of the current scope and go to step 4,
4. For each child scope start from step 2, treating each child as current scope till all the nodes in the tree are covered or till an error is raised in step2.

To Find the conflicts in parent scope, we use a method of the current scope 'LookupParent()' that follows the parent chain of scopes starting with current scope until it finds a scope where Lookup(name) returns a non-nil object.

The working of algorithm can be illustrated by the following figure

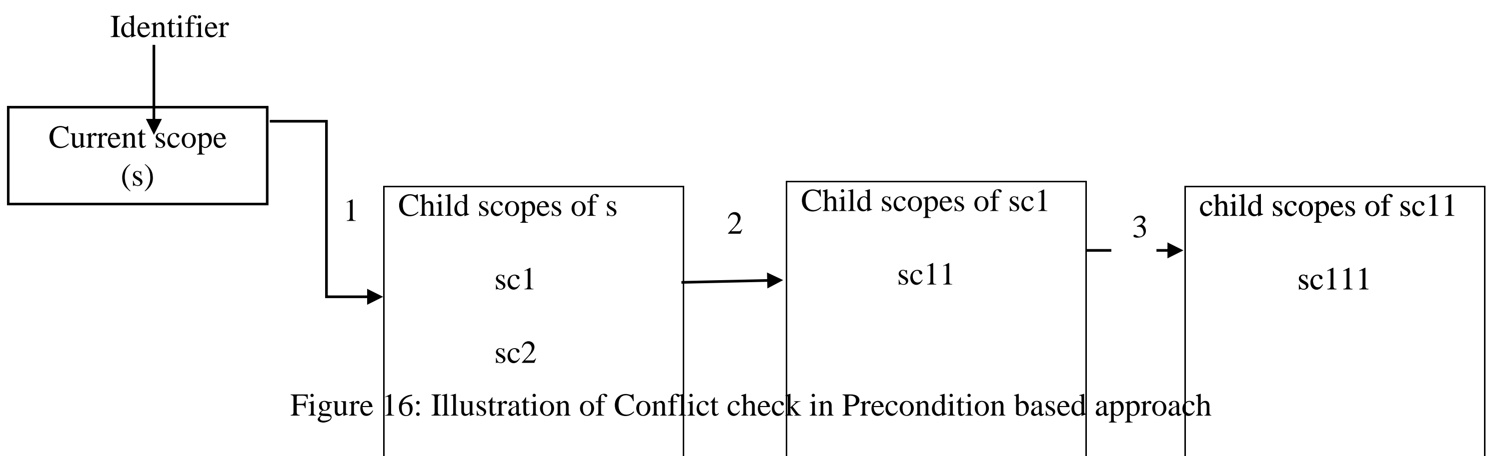



Figure 16: Illustration of Conflict check in Precondition based approach

**Time Requirements – Reference based approach**

In the reference based approach we collect the references of identifiers before applying the transformation, and after applying the transformation and compare them to see if the binding structure is preserved.

Algorithm

1. Sort the packages and files in each package
2. Inspect the files in each sorted package and collect the references of all identifiers in to an array or slice
3. After applying the transformation repeat steps 1 and 2
4. Create a map that maps all old references to new references which is empty initially and use it to find the differences in binding before and after the transformation

Though this algorithm looks simple to implement its time requirements are high in any average case programs.

Clearly reference based approach has higher time requirements.

**Space Requirements - Precondition based approach vs References based approach**

In the pre-condition based approach we search for the conflict in parent scope, current scope and child scopes. Each scope has one map that is used to figure out the conflict in that scope. Hence we need a data structure that stores the identifier references in related parent/child scopes. But in

The references based approach we need data structures to store the references of all identifiers in the program before and after transformation. Hence clearly the space requirements of the references based approach is more than pre condition based approach.

### 7.2 Empirical Analysis

We did an empirical study to find the occurrence of  corner case ( shadowing variable) in Go ,which is the primary reason for developing more sound and low performance reference based rename refactoring.

We did the empirical study on top 100 open source Go projects on Git Hub which constitute 4 million lines of code. The results explored many interesting facts about shadowing and reasons why Go programmers get trapped in to it in most of the cases.

Figures 17 and 18 shows the information related to corpus and results of the study.

| Projects | Packages | Files | Identifiers | LOC |
| --- | --- | --- | --- | --- |
| 100 | 181 | 2429 | 983240 | 4000000(approx.) |

Figure 17: Corpus

| Total Identifiers | Total Conflicting Identifiers | "err" Conflicting Identifiers | "ok" Conflicting Identifiers | Multiple Shadowing |
| --- | --- | --- | --- | --- |
| 983240 | 1091 | 278 | 68 | 289 |

Figure 18: Empirical Analysis- Shadowing variables

| %conflicting identifiers in total identifiers | % "err" identifiers in total conflicting identifiers | % "ok" identifiers in total conflicting Identifiers | % multiple shadowing in total conflicting identifiers |
| --- | --- | --- | --- |
| 0.1% | 25% | 6% | 26% |

Figure 19: Percentage Results of Empirical Analysis - Shadowing variables

From the results in figures 18 and 19 it is clear that frequency of shadowing is very low in Go. As low as 0.1 % of identifiers of the total identifiers tested are shadowed and among them ‘err’ and ’ok’ identifiers which are typical go style variable used to keep track of errors and boolean values account to 31 %.

Multiple shadowing in the above tables mean number of times a variable with same name is being shadowed in different scopes. Example below illustrates it clearly

In the example below global variable ‘largescope’ is shadowed in the function main nested if blocks and local variable ‘largescope’ in the outer if block in function main is again shadowed in inner or nested if block, Thus there are two different shadowings of the same variable in different scopes so we increase the multipleshadowings by 1 .

```
var largescope = ":-("   -->  //is shadowed below

func main() {
   largescope = ":-)"  // Don't change this

        if true {
           var largescope = "ds"     //is shadowed below
             fmt.Println(largescope)

                    if true {
                          var largescope = "ds"   // NOT SHADOWED
                        fmt.Println(largescope)

                    }
      }
}
```

Thus after understanding the meaning of multiple shadowings we know that only 74% of total conflicting identifiers are unique and most of the multiple shadows are of identifiers ‘err’ and ‘ok’. Our research also found that usage of short assignment operator ‘:=’ also may lead to accidental cases of shadowing by Go programmers [20].

In Go ‘err’ and ‘ok’ are most commonly used identifiers to store error messages and Boolean values returned by called method. Go supports multiple assignment statements like the one shown below

```
v, ok := a[x]
```

Where a is a map data structure and x is a key. a[x] gives the values stored with particular key x in the map. But the syntax above with the help of multiple assignment statement helps to find if the key x is actually present in the map or not. If x exists in map ok is true else ok is false.

In the empirical study from the corpus , the maximum number of conflicting identifiers in a single file was 23 and out of them 13 are ‘err’ , 2 or ‘ok’ and it had 16 multiple shadowing cases. We found the maximum conflicting identifiers in this github.com/coreos/rocket/Godeps/_workspace/src/github.com/cznic/ql/ql.go file and here are couple of snippets from the file that shows the identifier ‘err’ being shadowed. The symbols “…” indicate related code but which is not important to depict the shadowing.

**Snippet #1**

```
func (r *groupByRset) do(ctx *execCtx, onlyNames bool, f func(id interface{}, data
[]interface{}) (more bool, err error)) (err error) {
	t, err := ctx.db.store.CreateTemp(true)
	if err != nil {
		return
	}
	…
	…
	…
	ok := false
	…
	if err = r.src.do(ctx, onlyNames, func(rid interface{}, in []interface{}) (more bool, err
error) {
		if ok {
			…
			h0, err := t.Get(k)
			if err != nil {
				return false, err
			}

			…
			nh, err := t.Create(append([]interface{}{h, nil}, in...)...)
			if err != nil {
				…
			}
			…
			if err != nil {
				return false, err
			}
		}
…
	if onlyNames {
		_, err := f(nil, []interface{}{flds})
		return err
	}
	it, err := t.SeekFirst()
	if err != nil {
	…
	}
```

**Snippet #2**

```
func (r *distinctRset) do(ctx *execCtx, onlyNames bool, f func(id interface{}, data []interface{})
(more bool, err error)) (err error) {
	t, err := ctx.db.store.CreateTemp(true)
	if err != nil {
		return
	}
	…
	ok := false
	if err = r.src.do(ctx, onlyNames, func(id interface{}, in []interface{}) (more bool, err
error) {
		if ok {
			if err = t.Set(in, nil); err != nil {
				return false, err
			}

			return true, nil
		}

		flds = in[0].([]*fld)
		ok = true
		return true && !onlyNames, nil
	}); err != nil {
		return
	}

	if onlyNames {
		_, err := f(nil, []interface{}{flds})
		return noEOF(err)
	}

	it, err := t.SeekFirst()
	if err != nil {
		return noEOF(err)
	}
…
```

Here are snippets from the file

github.com/coreos/rocket/Godeps/_workspace/src/github.com/cznic/ql/ql.go that shows

identifier ‘ok’ being shadowed.

**Snippet #3**

```
func (r *whereRset) tryUseIndex(ctx *execCtx, f func(id interface{}, data []interface{}) (more
bool, err error)) (bool, error) {
	//TODO(indices) support IS [NOT] NULL
	c, ok := r.src.(*crossJoinRset)
	if !ok {
		return false, nil
	}

	tabName, ok := c.isSingleTable()
	if !ok || isSystemName[tabName] {
		return false, nil
	}
```

**…**

**Snippet #4**

```
func (r *whereRset) do(ctx *execCtx, onlyNames bool, f func(id interface{}, data []interface{})
(more bool, err error)) (err error) {
	//dbg("====")
	if !onlyNames {
		if ok, err := r.tryUseIndex(ctx, f); ok || err != nil {
			//dbg("ok %t, err %v", ok, err)
			return err
		}
	}
 ...

	ok := false
	return r.src.do(ctx, onlyNames, func(rid interface{}, in []interface{}) (more bool, err
error) {
		if ok {
			for i, fld := range flds {
				if nm := fld.name; nm != "" {
					m[nm] = in[i]
				}
			}
			m["$id"] = rid
			val, err := r.expr.eval(ctx, m, ctx.arg)
			if err != nil {
				return false, err
			}
			...
			x, ok := val.(bool)
			if !ok {
				return false, fmt.Errorf("invalid WHERE expression %s (value of
type %T)", val, val)
			}

		...
		}

		...
		m, err := f(nil, in)
		return m && !onlyNames, err
	})
}
```

Here is one more snippet from the file

/home/vzb0010/testcode/src/github.com/coreos/rocket/Godeps/_workspace/src/github.com/cznic/ql/stmt.go
Showing the shadowing cases of err and ok. This file has total 12 conflicting identifiers out of which 3 are 'err' and 2 are 'ok'.

**Snippet #5**

```
func (s *updateStmt) exec(ctx *execCtx) (_ Recordset, err error) {
	t, ok := ctx.db.root.tables[s.tableName]
	if !ok {
		return nil, fmt.Errorf("UPDATE: table %s does not exist", s.tableName)
	}
    ...
...

	for h := t.head; h != 0; h = nh {
		// Read can return lazily expanded chunks
		data, err := t.store.Read(nil, h, t.cols...)
		if err != nil {
			return nil, err
		}
          ...
        ...
		if expr != nil {
			val, err := s.where.eval(ctx, m, ctx.arg)
			if err != nil {
				return nil, err
			}
...
			x, ok := val.(bool)
			if !ok {
				return nil, fmt.Errorf("invalid WHERE expression %s (value of type %T)", val, val)
			}
...
...
		for i, asgn := range s.list {
			val, err := asgn.expr.eval(ctx, m, ctx.arg)
			if err != nil {
				return nil, err
			}
...
```

Thus from the above examples and results easy assignment syntax with the help short assignment operator (:=) and the feature of Go programming language to support multiple assignments in one statement and habit of go programmers to use typical Go style variables are the reasons for wide occurrences of shadowing for this two variables.

**7.3 Conclusion**

Rename refactoring is the most famous and widely used refactoring. Traditional approach to build automated rename refactoring is precondition checking an alternative approach that is sounder is proposed by Max Schäfer [9].

In this thesis we developed an automated rename refactoring for Googles Go programming language using the traditional precondition based approach and modified it to make it more robust using similar approach used by Max Schäfer [9]. We outlined the corner cases in Go that did not work in precondition checking approach and motivated the need for more robust tool.

We analyzed time/space requirements to show that precondition based approach though do not work in some shadowing cases, is more efficient in terms execution time. Our empirical analysis which included top 100 open source projects from Git Hub, showed that the frequency of occurrence of this corner cases is very low. Our literature survey gives us insight into how the professional programmers mostly use refactoring tools and follow coding standards by which we can believe that professional programmers will not usually commit silly mistakes of accidental shadowing.

As we now know that precondition based approach has better performance compared to references based approach and the reason for developing the references based approach which is the shadowing case in Go. The rational discussion in the above paragraph suggests that shadowing

cases will be very low in real world where refactoring tools are used, hence traditional precondition checking method of building refactoring tool would be an ideal approach for an efficient automated rename refactoring in Go

## References


1. Robert C. Martin, Clean Code: A Hand Book of Agile Software Craftsmanship. Prentice Hall, July, 2008.
2. Martin Fowler, Refactoring: Improving the Design of Existing Code. Booch Jacobson Rumbaugh, 2002.
3. E. Murphy-Hill and A. P. Black, Refactoring tools: Fitness for purpose. in IEEE Softw., 2008.
4. Emerson Murphy-Hill and Andrew P. Black, Why Don't People Use Refactoring Tools?. In Proceedings of the 1st Workshop on Refactoring Tools. ECOOP '07.
5. Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig, A Comparative Study of Manual and Automated Refactorings. In ECOOP '2013.
6. Gustavo Pinto and Fernando Kamei, What Programmers Say About Refactoring Tools? : An Empirical Investigation of Stack Overflow. In Proceedings of the ACM workshop on refactoring tools, 2013.
7. Florian Deißenb¨ock and Markus Pizka∗, Concise and Consistent Naming, In Proceedings of the 13th International Workshop on Program Comprehension, (IWPC'05).
8. Emergence of Go programming language Retrieved March 30, 2015 , from redmonk.com

   URL http://redmonk.com/dberkholz/2014/03/18/go-the-emerging-language-of-cloud-infrastructure/

9. Max Schäfer, Torbjörn Ekman and Oege de Moor. Sound and Extensible Renaming for Java. In Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and application, Pages 277-294, OOPSLA '08.
10. Venera Arnaoudova, Laleh M. Eshkevari, Massimiliano Di Penta, Member, Rocco Oliveto, Giuliano Antoniol, and Yann-Ga€el Gu_eh_eneuc. REPENT: Analyzing the Nature of Identifier Renamings. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 40, Issue NO. 5, Pages 502 - 532, MAY 2014.
11. Maartje de Jonge and Eelco Visser. A Language Generic Solution for Name Binding Preservation in Refactorings. In Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, Article No. 2, LDTA '12.
12. Asger Feldthaus and Anders Møller. Semi-Automatic Rename Refactoring for JavaScript. In Proceedings of the ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA), Pages 323-338, 2013.
13. History of Go Retrieved March 30 , 2015, from Wikipedia
URL http://en.wikipedia.org/wiki/Go_%28programming_language%29
14. Features of Go Retrieved March 30, 2015, from Go Official Website
URL https://golang.org/doc/
15. Information about Go ast package Retrieved March 30, 2015, from Go Official Website
URL http://golang.org/pkg/go/ast/
16. Abstract Syntax Tree in general Retrieved March 30, 2015, from Wikipedia
URL http://en.wikipedia.org/wiki/Abstract_syntax_tree
17. Go/Loader Description Retrieved March 30, 2015, from Wikipedia

URL https://godoc.org/golang.org/x/tools/go/loader

18. Jeffrey L. Overbey and Ralph E. Johnson. Differential Precondition Checking: A Lightweight, Reusable Analysis for Refactoring Tools. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, Pages 303-312
19. YangSun Lee[1] and YunSik Son[2]. A Study on Verification and Analysis of Symbol Tables for Development of the C++ Compiler. In Proceedings of the International Journal of Multimedia and Ubiquitous Engineering. Vol. 7, No. 4, October, 2012
20. Accidental shadowing in Go Retrieved April 7, 2015 , from Wordpress

    URL https://developmentality.wordpress.com/2014/03/03/go-gotcha-1-variable-shadowing-within-inner-scope-due-to-use-of-operator/
21. Keywords in Go programming language Retrieved March 30, 2015 , from Go official website

    URL https://golang.org/ref/spec#Keywords
22. Pre declared identifiers in Go programming language Retrieved March 30, 2015 , from Go official website

    URL https://golang.org/ref/spec#Predeclared_identifiers