

An Extract Function Refactoring for the Go Language

by

Steffi Mariya Gnanaprakasa Paul Arasu

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
August 1, 2015

Keywords: Refactoring, Extract Function, Live Variable Analysis

Copyright 2015 by Steffi Mariya Gnanaprakasa Paul Arasu

Approved by

Jeffrey L. Overbey, Chair, Assistant Professor of Computer Science
Munawar Hafiz, Assistant Professor of Computer Science
Dean Hendrix, Associate Professor of Computer Science

Abstract

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal code structure without changing its external behavior [22]. Refactoring is specified as a parameterized program transformation that requires a set of preconditions to be satisfied.

The Go Doctor is a refactoring tool for the Go language. This thesis focuses on the design and implementation of an Extract Function refactoring adapted to the unique requirements of the Go language. The code that is to be refactored is parsed and type checked before the process is initialized. The tool ensures that certain preconditions are met; if they are not, it warns the user with the exact details as to why the refactoring will not continue. Then, a Live Variable analysis is used to determine what variables need to be passed to, returned from and declared in the new function that is created from the extracted code.

To analyze the robustness of the refactoring, it was applied to 200 random statements selected from the top 100 GitHub projects using Go. These projects were also used to assess the potential impact of some of the refactoring limitations - specifically, its inability to extract *return* statements, *defer* statements, and anonymous functions.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my mentor Dr. Jeffrey L. Overbey for his diligent support and guidance. His enduring motivation has fostered in me, a never-ending opportunity to excel during the course of this research. Without his persistent help this thesis would not have been possible.

I would also like to thank Dr. Munawar Hafiz, for his valuable inputs on this research and accepting to be a part of my advisory committee. I owe much gratitude to Dr. Dean Hendrix for being a part of my advisory committee. I thank all the committee members for reviewing this manuscript and providing their valuable feedback.

This research work wouldn't have been enjoyable if it wasn't for the support of my colleagues. Thanks to all my friends for their constant support and making my time at Auburn a memorable journey.

Finally, I would like to thank my parents, Grace Mary and Paul Arasu, and my brother Tony Jefferson for their unconditional love and support. Thank you for all the encouraging words and for believing in me. I thank God almighty for all the blessings that He has bestowed upon me and for guiding me all the way.

Table of Contents

| | |
|--|-----|
| Abstract | ii |
| Acknowledgments | iii |
| List of Figures | vi |
| List of Tables | ix |
| 1 Introduction | 1 |
| 1.1 About the language | 2 |
| 1.2 Design of Go | 3 |
| 1.3 The Go Doctor | 4 |
| 1.4 Thesis Organization | 6 |
| 2 Related Work | 7 |
| 2.1 Refactoring | 7 |
| 2.2 Identifying Refactoring Candidates | 8 |
| 2.3 Behavior preserving in Refactoring | 9 |
| 2.4 Types of Refactoring | 11 |
| 2.4.1 Rename Refactoring | 11 |
| 2.4.2 Extract Function Refactoring | 12 |
| 2.4.3 Reverse Conditional Refactoring | 13 |
| 2.4.4 Consolidate Duplicated Conditional Fragments | 13 |
| 2.4.5 Inline Method Refactoring | 13 |
| 2.4.6 Move Method Refactoring | 14 |
| 2.4.7 Create Abstract Superclass | 14 |
| 2.5 Most Commonly used Refactoring | 14 |
| 2.6 Function Outlining | 15 |

| | | |
|-------|--|----|
| 2.7 | Extract Function Refactoring | 16 |
| 2.7.1 | Identification of Extract Function opportunities | 16 |
| 2.8 | Refactoring Tools | 17 |
| 3 | Implementation | 20 |
| 3.1 | Refactoring Infrastructure Framework | 20 |
| 3.2 | Extract Function Refactoring | 22 |
| 3.2.1 | Precondition Check | 23 |
| 3.2.2 | Live Variable Analysis | 28 |
| 3.2.3 | Live Variable Analysis in Detail | 31 |
| 3.2.4 | Implementation of the Dataflow analysis in Go Doctor | 33 |
| 4 | Evaluation | 40 |
| 4.1 | Purpose | 40 |
| 4.2 | Testing Framework | 40 |
| 4.2.1 | Go Patient | 42 |
| 4.3 | Test Code | 42 |
| 4.4 | Testing Results | 43 |
| 4.5 | Decision made based on Results | 44 |
| 4.5.1 | Return Statements | 44 |
| 4.5.2 | Anonymous Functions | 46 |
| 4.5.3 | Defer Statements | 47 |
| 5 | Conclusion and Future Works | 50 |
| | Bibliography | 52 |
| A | Toggle Variable Refactoring | 57 |
| A.1 | Precondition Check | 57 |
| A.2 | Working | 57 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Selecting Code for Extract Function Refactoring | 5 |
| 1.2 | Initiating Extract Function Refactoring | 5 |
| 1.3 | Program after Transformation | 6 |
| 2.1 | Reverse Conditional Refactoring | 13 |
| 3.1 | Working of Extract Function Refactoring | 21 |
| 3.2 | Package Diagram of Go Doctor | 22 |
| 3.3 | A Go Language Source Code | 23 |
| 3.4 | Nodes of Path Enclosing Interval of highlighted statement in Figure 3.3 | 23 |
| 3.5 | Path Enclosing Interval of highlighted statement in Figure 3.3 | 24 |
| 3.6 | An Example for Defer Statement | 25 |
| 3.7 | Output of Program in Figure 3.6 | 25 |
| 3.8 | Example of Anonymous function | 25 |
| 3.9 | Extract Function Refactoring on a labeled branch statement <i>break</i> | 27 |
| 3.10 | Extract Function Refactoring on Branch Statement <i>break</i> | 29 |
| 3.11 | Control Flow Graph for Program in Figure 3.10 | 30 |

| | | |
|------|---|----|
| 3.12 | Variables Defined | 34 |
| 3.13 | Variables Defined and Assigned | 34 |
| 3.14 | Extraction of Partially Assigned Variables | 34 |
| 3.15 | Illustration of Partially Assigned Variables after Extraction | 35 |
| 3.16 | Structure Declaration by Value | 35 |
| 3.17 | Function Call for Structure Passed by Value | 35 |
| 3.18 | Structure Declaration by Reference | 36 |
| 3.19 | Function Call for Structure Passed by reference | 36 |
| 3.20 | Variables Used | 36 |
| 3.21 | Example Program for Function Call Replacement | 39 |
| 3.22 | Output Program Function Call Replacement | 39 |
| 4.1 | Graphical Report for Tests | 43 |
| 4.2 | Proof for Decisions | 45 |
| 4.3 | Return Statement Usage | 46 |
| 4.4 | Program with Return Statement | 46 |
| 4.5 | Program with Extracted Return Statement | 47 |
| 4.6 | Program with Return Statement that can be Extracted | 47 |
| 4.7 | Program with Extracted Return Statement from Figure 4.6 | 48 |

| | | |
|-----|---|----|
| 4.8 | Defer Statement Usage | 49 |
| 5.1 | Removing Local Variable Declaration | 50 |
| A.1 | Multivalued Function | 58 |
| A.2 | Multivalued Function returns of the Same Type | 58 |
| A.3 | Multiple Expression Declaration | 58 |

List of Tables

| | |
|----------------------------------|----|
| 4.1 List of Test Cases | 44 |
|----------------------------------|----|

Chapter 1

Introduction

According to Fowler [7], Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. The main purpose of refactoring is to make the code more reusable and easier to understand and the refactoring that needs to be executed is specified as parametrized program transformations that require a set of preconditions to be satisfied. If the preconditions are met and if the transformation does not alter the behavior of the code then the transformation is applied [4].

Refactoring is a cumbersome process when the programmer has to perform them manually and hence it is necessary that there are automatic refactoring tools integrated with the development environment to help programmers refactor their code. Refactoring tools aren't fully automatic, instead they are directed by the user who determines what refactoring to perform and input the parameters to perform the refactoring[39][43][40]. For instance, if the programmer decides to use rename refactoring that makes it easier to rename identifiers and contexts where they are used inside the code, the user has to enter the identifier to rename as well as the new name of the identifier. In the case of a Extract Function refactoring, where the programmer can extract a contiguous lines of code from a long function and create a new function with the extracted code, the user has to enter a name for the new function.

Some of the problems faced by the programmers who use refactoring tools for Extract Function refactoring are listed below [45],

- They have difficulty determining the types of language constructs that must be selected.
- They face difficulty while identifying what the error thrown by the refactoring tool is really about.

- They have difficulty handling conditional branching statements that are a part of the statements that are to be refactored.
- When the refactoring tools move the declarations inside the new function, and this can alter the behavior of the code.

Some of the refactoring tools available in market are, IntelliJ IDEA[40] and a number of refactoring tools that are built into the Eclipse JDT and CDT[44]. Tools like JUNGL[39] a refactoring tool, were developed for research purpose.

1.1 About the language

Go Programming Language was made an open source project in November 2009, developed by Robert Griesemer, Rob Pike and Ken Thompson. Go is a compiled, concurrent, garbage collected, statically typed language developed at Google. Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. Some of the features of Go Language that makes it more efficient and user friendly when compared to other languages are[65],

- The dependency management(handling how importing other files into the current program) of Go makes Go compilations faster than C or c++ Compilations(that includes all the files and libraries).
- The dependency graphs have no cycles, since there can be no circular imports in the graph.
- Unlike traditional C, C++ and Java models, Go includes linguistic supports for concurrency, garbage collection, interface types, reflection and type switches.

- Go has no explicit memory-freeing operation, the allocated memory returns to the pool using garbage collector, unlike C or C++ where too much of programming efforts is required for memory allocation and freeing.
- Automatic garbage collections in Go, makes concurrency and multithreaded programming code easier to write[66].

1.2 Design of Go

Go Language, an object oriented programming language with C-like syntax is designed as a systems language and focuses on concurrency and multiprogramming. It supports a mixture of static and dynamic typing making it more safe and efficient. Go contains objects and structs instead of classes. Although there is no explicit subtype declarations, sometimes structural subtypes are inferred. Go Language supports pointer types by putting an asterisk in front of the type but does not allow pointer arithmetic (C allows pointer arithmetic)[20].

- Objects and Methods - Like in C++, methods in Go Language are defined outside the struct declarations and the receiver must be explicitly named. Go Language allows multiple return values although it does not support overloading or multiple methods.
- Embedding instead of Inheritance - Reuse in Go Language is supported by embedding (including an anonymous field in a struct is known as embedding making all of the methods of the embedded fields made accessible to the struct[21]). If a method or field cannot be found in an object's type definition, then the embedded types are searched and method calls are forwarded to the embedded object.
- Interfaces - They are abstract representations of methods. If an object implements the methods of an interface then it has that interface in its type. By default every object implements a empty interface *interface*.

- Functions and Go routines - Go Language supports functions(methods without receivers), function pointers and closures. Go Language routines are functions that execute in parallel with other go routines. The keyword *go* in front of a function call executes a function as a go routines. Communications between go routines are done by the concept of message passing which is implemented by *channels*.
- Object Initialisation - Objects are created from the struct definition with or without the *new* keyword. When an new object is created fields are initialized by the user or they are set to default values. Go Language does not have constructors.
- Packages - The source code in Go Language is structured inside packages and Go Language provides two scopes of visibility,
 - Members beginning with lowercase are only accessible inside their package.
 - Members beginning with an upper case letter are publicly visible.

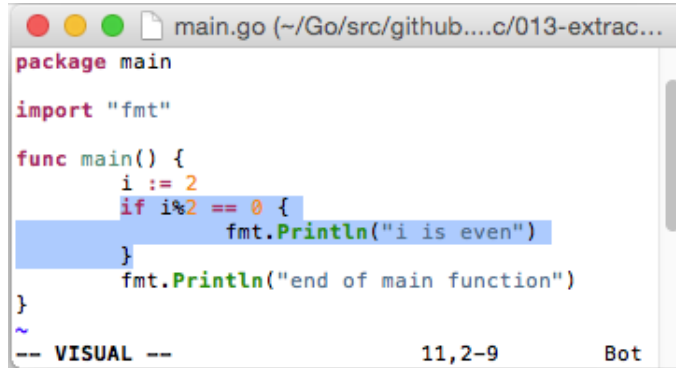
1.3 The Go Doctor

Go Doctor, a Go Language refactoring tool implements few of the most commonly used refactorings like the Rename refactoring, the Extract Function refactoring, Toggle Variable refactoring and Extract Local Variable refactoring. This thesis explains how the Extract Function refactoring in Go Doctor is implemented and the different analysis required to implement the refactoring. Figure 3.1 explains how the tool works specifically for Extract Function refactoring.

The refactoring engine is the entry point for Go Doctor which lists all the available refactoring to the user. The engine calls the refactoring interface, which contains all the refactoring. The Figure 3.2 describes the package diagram of how the different packages within Go Doctor is organised.

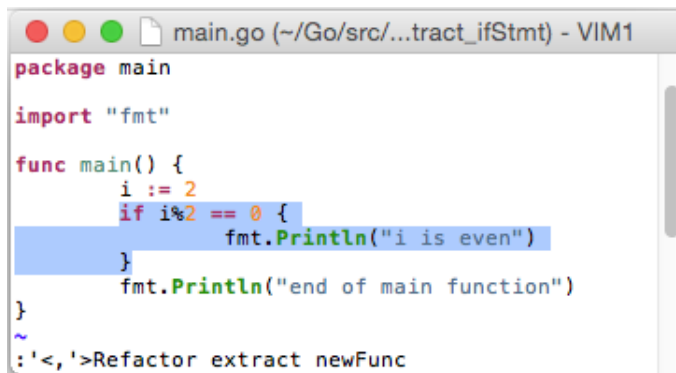
The user selects the set of statements they want to extract and selects the Extract Function refactoring and enters a valid function name. The code on which refactoring is

performed on is parsed and type checked and then the Extract Function refactoring is initiated. The Go Doctor makes sure that the preconditions are met so that the refactoring is behavior preserving[23]. After which dataflow analysis is performed on the code and the code is transformed.



```
package main
import "fmt"
func main() {
    i := 2
    if i%2 == 0 {
        fmt.Println("i is even")
    }
    fmt.Println("end of main function")
}
-- VISUAL --          11,2-9      Bot
```

Figure 1.1: Selecting Code for Extract Function Refactoring

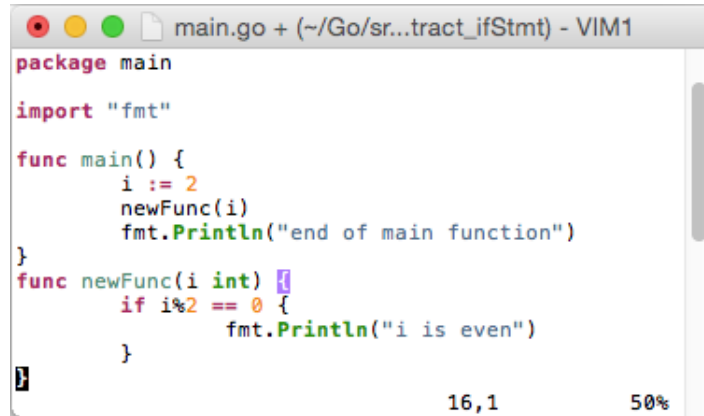


```
package main
import "fmt"
func main() {
    i := 2
    if i%2 == 0 {
        fmt.Println("i is even")
    }
    fmt.Println("end of main function")
}
~
: '<', '>Refactor extract newFunc
```

Figure 1.2: Initiating Extract Function Refactoring

The Vim version of the steps involved in Extract Function refactoring is explained in Figure 1.1, Figure 1.2 and Figure 1.3.

1. Code Selection - The code selected should be a part of a block statement (should be inside a function). The selection has to be a contiguous set of statements.

The image shows a screenshot of a Vim editor window. The title bar reads "main.go + (~/Go/sr...tract_ifStmt) - VIM1". The code displayed is as follows:

```
package main

import "fmt"

func main() {
    i := 2
    newFunc(i)
    fmt.Println("end of main function")
}

func newFunc(i int) {
    if i%2 == 0 {
        fmt.Println("i is even")
    }
}
```

The status bar at the bottom right shows "16,1" and "50%".

Figure 1.3: Program after Transformation

2. Initialize Refactoring - The refactoring engine trigger the Extract Function refactoring after the *extract* keyword followed by the user specified name for the new function is entered.
3. Code after Transformation - After Extract Function refactoring was implemented, the code has been transformed preserving the behavior of the code.

1.4 Thesis Organization

- Chapter 2 proposes the concept of refactoring to improve the software quality and familiarizes the reader with the concept of Extract Function refactoring which started off with they concept of function outlining in compiler. Explains the different refactoring tools
- Chapter 3 describes the refactoring infrastructure framework for Go Doctor, explains the implementation the Extract Function refactoring and gives an overview of the testing framework used.
- Chapter 4 evaluates the Extract Function refactoring by testing it over 4 million lines of code and argues the reasons why certain decisions were taken.
- Chapter 5 discusses the future work that can be done based of the current work.

Chapter 2

Related Work

Software Engineering when viewed from a historical perspective had its developments over decades of integrating software into the daily operations of various institutions, planning and controlling software projects due to costs and schedule delays[1].

According to Ronan Fitzpatrick[2], Software Quality refers to the implementation of industry-defined set of desirable features into a product that is more efficient, effective and gives produces reliable performance. Software Inspection, a method used to improve the software quality involves code walk through, where the code is examined so that every piece of logic is covered and every branch or case is considered. It also makes sure all the required functionality is implemented and the errors are caught[3]. Code smells are metaphors used to describe poorly designed code structure and bad programming practices, that may lead to bugs that costs a lot of time and effort to identify[4]. There are researches for developing automated tools that assure software quality by detecting code smells in the source code in languages like Java [5][6]. Code smells when removed during the earlier phase of development cycle can be more cost and time effective. Code smells can be avoided while writing the program but there can be certain times when the programmer may not be aware of it. In those cases, it is recommended to use refactoring to eliminate code smells.

2.1 Refactoring

According to Martin Fowler[7],Refactoring refers to changes made to the internal structure of software such that the modified software code is more comprehensible without changing the observable behavior of code. Refactoring is a technique that is used to clean up code in a more efficient and controlled manner so as to minimize bugs. There has been a few

researches that study the effects of refactoring on errors found in the program[8] and how refactoring affects the quality of the software[9]. When programmers change the code, the structure of the code gets distorted and loss of structure can cause a cumulative effect. The more cluttered the code gets, the more difficult it is to fix the code. Refactoring is the process of clarifying and simplifying the design of the existing code, without changing its behavior. In agile software development, programmers maintain and extend their code from iteration to iteration and without continuous refactoring, the quality of the code can depreciate leading the code to rot (unhealthy dependencies between classes and package, bad allocation of responsibilities to classes and functions, duplicate code) that spreads and worsens the code quality. Refactoring code prevents rot and makes the code easy to maintain and extend (by adding new features and implementing bug fixes) [10].

2.2 Identifying Refactoring Candidates

Once the developer discovers the perfect situation to do refactoring, he needs to identify the potential candidates in the code to refactor. In order to determine the prospective candidates for refactoring, semantic analysis of the code is required.

Kataoka et al [11] proposed a method for identifying refactoring candidates by using program invariants. If the program invariant matches the invariant pattern that already exists at a particular point in the program, it indicates the applicability of a specific refactoring. The author also talks about how their basic approach is independent as to static or dynamic analysis of how invariants are found. The static analysis (Human Analysis) requires the programmer to elucidate his program with respect to the design requirements so that the implicit invariants can be identified and this can be cumbersome. An alternative approach was to automatically infer invariants that is done statically or dynamically. With dynamic approach, the program is trained to trace variables of interest and the result of the dynamic approach depends on the quality of test suites that is used to infer the invariants. There

has been more studies on the static and dynamic invariant inference tools by Nimmer and Ernst[12].

Another study[13] suggests an approach that identifies the refactoring candidates by using dependency graphs. This techniques identifies classes that can be potential refactoring candidates where it identifies or searches for dependency cycles in long classes which makes these classes difficult to understand, test and reuse. There are a few researches who use metrics to identify refactoring candidates[14, 15, 16].

There are many tools that can be used to identify code smells in programming languages like Java JLint[17] and the C analyzer Lint[18] that support automatic code inspection. JDeodorant[19] is another tool that looks for code smells and identifies God Classes (an object that controls many other objects in the system), suggests where to apply EXTRACT CLASS refactoring and helps the programmer perform the refactoring in an effective manner.

When a developer is in the process of writing code, sometimes they are not aware of the situation that they are manually refactoring until halfway through their code modification process, this is called *late awareness dilemma* which contributes to the refactoring tools under use. A novel refactoring tool called Benefactor[42] try to reduce this by detecting the manual refactoring that is performed by the developer and suggests the automatic refactoring options.

2.3 Behavior preserving in Refactoring

Refactoring refers to structural changes made to the program such that these changes preserve the behavior or the semantics of the code. Although there are works [22] done to verify how refactoring could improve the program design and structure, we would also need to focus on the semantics preserving of the program.

Fowler[7] discusses the concept of self-testing code (practise of writing comprehensive tests in conjunction with software), where he had to write the expected output from the test code and compare it with the transformed code and check if they are the same if not,

it indicates that there is a bug in the code transformation. The purpose is to make sure that code that is transformed matches the expected output written by the programmer. The author suggests writing a suite of tests including boundary conditions and focus on the areas where things might go wrong in order to detect the bugs that may gradually appear over time when additional features are added. Creating a test suite is the best means to find bugs using regression testing. This approach is not fool-proof because testing the code solely based on the output is not sufficient. Opdyke[23] came up with a few properties for behavior preservation of refactored code,

- Unique Superclass - Post refactoring the class must have at most one superclass and this superclass must not be one of its subclasses.
- Distinct Class Names - Post refactoring each class must have a unique name.
- Distinct Member Names - The members and variables in a the refactored class must have unique name.
- Inherited Member Variables not Redefined - The member variable that is inherited from the superclass must not be redefined in the sub class.
- Compatible Signatures in Member Function Redefinition - Post refactoring if the member function of a function in the superclass is redefined in the subclass, it has to be compatible.
- Type-Safe Assignments - Post refactoring the type of the expression assigned to a variable must be the same as the type the variable was declared with.
- Semantically Equivalent References and Operations - This states that the versions of the code before and after the refactoring produce semantically equivalent references and operations.

According to Tom Mens et al[67], depending on domain or user-specific concerns there can be a wider range of behavior that may or may not be preserved in refactoring,

- *Real-time Software* - The crucial aspect of behavior is the execution time of certain sequence of operations and requires the refactoring to preserve all the temporal constraints (time related factors of the software).
- *Embedded Software* - The crucial aspect of behavior that needs to be preserved by refactoring are the memory constraints and power constraints.
- *Safety-critical Software* - There are concrete behaviors of safety (such as liveness, stability, reliability of the system) that needs to be preserved by refactoring.

The author suggests that dealing with behavior preservation in a pragmatic way is with the help of rigorous testing and ample number of test cases that passes after refactoring gives a good evidence that the refactoring is behavior preserving, but there are also cases when the tests rely on the program structure that is modified causing certain refactoring to invalidate the tests. The author gives another approach for behavior preservation by adopting a weaker notion of behavior preservation that is insufficient to formally guarantee the full preservation of program semantics.

2.4 Types of Refactoring

2.4.1 Rename Refactoring

One of the most commonly used refactoring among all the available refactorings. This refactoring is used by the programmer who wants to change the name of an identifier to another name (that is not a keyword) that is more appropriate for it.

- When the name of an identifier is changed, then all references to that identifier must be changed to new name.
- Naming Analysis - When a programmer tries to rename a method that is a part of an interface, it may affect all the classes that implement that interface in different packages too. Hence the programmer uses reflexive transitive closure[24] to get a list

of references to the method that is to be renamed, so that the behavior of the code is not changed.

2.4.2 Extract Function Refactoring

According to Clean Code [68], a function has to be small; perform just one task that is not repetitive; must have a descriptive name. To make the function do just one task, the programmer needs to make sure that all the statements within the function are of the same level of abstraction. It is always advisable to limit a method to one functionality for better design, the effect of Extract Function refactoring is improving the code maintainability. Some of the things that need to be taken care of while performing Extract Function refactoring are,

- The newly extracted method must have access to all local variables it uses and must return those local variables that would be used by the code following the extracted block.
- Branch statements and exit statements like return must be handled in an effective manner if they are a part of the extracted code.
- Extracting code that has more than one assignment statement cannot be extracted since the method can return only one variable.
- Live variable analysis[25] is used to eliminate those definitions of the variables that are not used in the program. This is also used to identify those variables that need to be passed, returned and declared inside the extracted code. Live variable analysis is mainly used to determine the variable definitions inside a loop. If the variable is redefined inside a loop and used in statements following the extracted code, it must be returned.

2.4.3 Reverse Conditional Refactoring

This refactoring is done to improve the readability of the code. When the programmer comes across conditional statements such as,

```
if(condition){
    //statements
}else{
    //statements
}
```

Figure 2.1: Reverse Conditional Refactoring

The programmer may want to improve the readability of code by swapping the ‘if’ and the ‘else’ part of the code. This transformation would require the conditions of the original ‘if’ statement to be reversed. This involves inverting the condition that is in the original ‘if’ statement, switch the clauses interchange the statements under the ‘if’ and ‘else’ statement and test for behavior preservation[7].

2.4.4 Consolidate Duplicated Conditional Fragments

This refactoring can be applied whenever there occurs a situation where the same fragment of code is in all branches of the conditional expression such as ‘cases’ in the type switch statements. Refactoring this kind of design defects is called code hoisting, an optimization to reduce the code size by removing those statements that occur in multiple code paths from a single common point in Control Flow Graph(CFG). We need to make sure that the code inside the branches and the duplicated code fragments are independent of each other.

2.4.5 Inline Method Refactoring

Method inlining is basically the inverse of method extraction. When a programmer identifies a method that is too small to be an independent method, the programmer may want to merge it with another method. Those variables that are local to the inline method

must become a part of the target method now; hence the naming conflicts between the variables of the inline method and the target method must be resolved which may otherwise cause a compilation error. This might change the behavior of the code if not used carefully, it can change the access specifier of the inline method to that of the target method.

2.4.6 Move Method Refactoring

This refactoring is performed if the programmer thinks that the method defined in a particular class does not fit in and needs to be moved to a class where it would be a more apt function. This involves more collaboration between classes and there are changes that needs to be made in the target class so that it can accommodate the new method that is being moved into it making both the classes highly coupled.

2.4.7 Create Abstract Superclass

This refactoring is used when there is a duplicate code in both classes, making that part of code common among the two classes, hence there can be an inheritance structure created from this scenario by creating a blank abstract superclass and then making the original classes subclasses of this superclass[27].

2.5 Most Commonly used Refactoring

Negara et al [28] studied the popularity of various refactorings, considering both manual and automated with the help of 23 participants, based on whether they used the automated refactoring option offered by the eclipse tool, or how they performed the refactoring manually. It was said that the programmers used 11% more manual refactoring than the automated one. It was also proved that developers who were less than 5 years in experience tend to perform 28% more manual refactoring than automated than those with experience. Developers with more than 10 years experience performed 49% more manual refactoring than automated refactoring, the reasons could be that the experienced programmers learnt how to refactor

even before the automated refactoring tools came into existence. Studies by Mohsen Vakilian et al [29] talks about some factors that may affect automated refactoring such as, awareness, trust, invocation method and major mismatch between what the programmer wants and what the automated tool does.

When talking about the most popular refactoring tools among the manual and the automated versions of refactoring. The top five most popular refactorings used were Rename Local Variable, Rename Method, and Extract Local Variable (these three were common in automated and manual refactoring), Extract Function (popular in manual), Rename Field(automated). Hence when building a refactoring tool, the tool builders must be more focused on building the popular refactorings first and make sure they are accurate.

They also studied the amount of time needed by these developers to perform refactorings, and it appeared that Extract Function refactoring consumed the max amount of time, in manual and automated refactoring, followed by extract local variable and rename class.

2.6 Function Outlining

According to Pettis and Hansen, in their work on code positioning[69], whose primary purpose was to reduce the overhead of instruction memory hierarchy. The main idea is to separate the frequently executed statements from the rarely executed statement in a program unit for the purpose of optimizing the code layout. Suganuma et al [70] explains the concept of region based compilation technique using dynamic compilation system, where the compiled regions are selected code portions without rarely executed code. The paper assumes that each method is represented as a control flow graph(CFG) with a single entry and single exit block. Peng Zhao et. al [71] describe function outlining as a technique that splits a region of code into an independent, new function and replaces that region with the function call to the new function that was created. Some of the positive impacts of function outlining is that it enables function inlining and improves code locality. With function outlining, new function

calls are introduced and this causes program control to jump between the original program unit and the outlined function.

2.7 Extract Function Refactoring

Extract Function refactoring helps decomposing large multifunctional methods into smaller methods that performs a single task, making re-usability possible when the methods are finely grained.

2.7.1 Identification of Extract Function opportunities

Nikolas Tsantalis and Alexander Chatzigeorgiou[33] studied some of the design flaws listed in Fowler et al ,1999 [22] that can be fixed with the help of Extract Function refactoring which are,

- **Duplicated Code**, a set of statements that perform a particular function and is repeated throughout the code, this code can be extracted and put into a single method to which a method call can be made.
- **Long Method**, long multifunctional methods are decomposed into methods that perform a single task using Extract Function refactoring.
- **Feature Envy**, a term used to describe a situation when a method is a part of a class but uses several data from another class. This method is extracted from the current class and placed inside the class from which it uses the data.

They came up with a set of methods to identify two main categories of Extract Function refactoring opportunities,

1. **Complete Computation Slice** returns those variables whose value is modified by assignment statements throughout the body of the original method. The algorithm takes in an input method M and identifies all the variables whose values are modified

within the method using at least one assignment statement covering the computation of the corresponding variable and returns a set of slice for extract refactoring.

2. **Object State Slice** returns object references (the local variables or fields of class containing the original method) that points to the objects whose states are affected by the method invocation that modifies at least one of its attributes. This algorithm takes as input a method M and returns a set of slice for each reference inside the method M that points to an object whose state is affected by at least one statement containing an appropriate method invocation or direct field modification.

2.8 Refactoring Tools

In 1997, Don Roberts and his research group developed the Smalltalk Refactoring Browser[34]. The Refactoring Browser is a tool that renders automatic support for many of the common refactorings in Smalltalk development. The earlier versions of Smalltalk Refactoring Browser was a stand-alone tool making it difficult for developers to use them. Some of the important features that needs to be considered when building a refactoring tool such that the programmers may frequently use it are,

- Integrate the refactoring tool into the standard development tools.
- The refactorings must be fast and reasonably correct.
- The refactoring tool must avoid purely automatic reorganization.

Another refactoring tool, JUNGL[39] focuses on making transformation to the source code of a program rather than any convenient intermediate representation of the code using scripting transformation. JUNGL introduced the idea of using a scripting language for refactoring. Making use of scripting languages for such transformations can be beneficial for functional features such as manipulating AST (Abstract Syntax Tree) and graphs more generally, logical queries for expressing complex relationships between program elements.

This language is implemented in a .NET platform. The main data structure is the program graph built from functions of the program, where each node and edge has a kind represented by string. Only if there is a successor to a node or statement, is there a lazy edge added from the current node to the next node.

In order to transform a program based on rename refactoring there are a certain set of conditions that needs to be checked,

1. Variable Binding
2. Conflicting declarations

To implement Extract Function Refactoring one needs to,

- Check Validity of the extracted code - Single Entry Single Exit criteria that follows the concept of nodes that dominate the other, it takes three parameters entry node to the method that contains the block, start node and end node of the block. The start node is said to dominate the end node if there is only path from the start node to reach the end node. Checking on the scoping information as well.
- Parameters to the new function - When trying to extract a code into a new function, the programmer needs to consider those parameters that are,
 - passed by value,
 - passed by reference,
 - output parameters that only return a result.
- Placing Declarations - Moving them inside or outside the block depending on their usage.
- Transforming - Inserting the new function call and replacing the extracted code with the function call to the new function.

Refactoring Rubicon for Extract Function refactoring can require some serious work, the method has to be analyzed and temporary variables need to be identified and placed in the proper positions so as not to change the behavior of the code. IntelliJ IDEA[40] was one of the first java IDE to cross-refactoring Rubicon. This consists of a parser and AST that converts the source code into a form of AST called *Program Source Interface (PSI)* that supports low level tree traversal and higher level semantic operations. Followed by building a CFG of the methods to be processed, which consists of nodes that are language independent instructions linked by edges specifying the possible variants of control flow graph.

Eclipse[41], an open source project is a generic development environment and has refactoring support and also performs some analysis on the code and supports various kinds of refactorings.

Chapter 3

Implementation

3.1 Refactoring Infrastructure Framework

The purpose of refactoring and the commonly used refactorings were discussed in the previous section, this section is dedicated to explain the implementation of the Extract Function Refactoring. Clean Code [68] states that a function has to be small; perform just one task that is not redundant; must have a descriptive name. To make the function do just one task, the programmer needs to make sure that all the statements within the function are of the same level of abstraction. It is always advisable to limit a method to one functionality for better design, hence programmers use Extract Function refactoring on long functions for improving the code maintainability. The main purpose of Extract Function refactoring is to decompose large code fragments into small cohesive functions, the user is asked to select the code and enter the new name for the new function that is to be created.

Go Doctor[60], a Go Language refactoring tool works on vim. Some of the refactorings that can be implemented by Go Doctor are Rename refactoring, Toggle Variable refactoring, Extract Function refactoring and Extract Local Variable refactoring. As explained in the introduction section, the below flow chart describes the generalized working of Extract Function Refactoring.

1. Get User Input - There has to be an interface that interacts with the user to get the lines selected by the user and the new name for the function, this is done by the *Command Line Interface (CLI)* and *Protocol* that provides the standard mechanism for text-editors to communicate with refactoring engines.

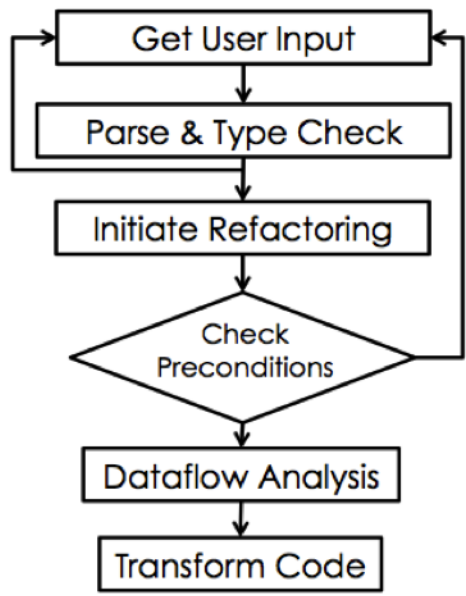


Figure 3.1: Working of Extract Function Refactoring

2. Parse and Type Check - The *FileSystem Interface* manages the files that are loaded and those files that are changed after the refactoring has been applied. This provides the ability to read files, load files, modify and remove files.
3. Initiate Refactoring - The *Engine* is the entry point for Go Doctor refactoring, all available refactorings must be given a unique name and this unique name must be added to the refactoring list by creating a refactoring object for it. For example,

Unique name - *extract*

Refactoring Object - *refactoring.ExtractFunc*

4. Refactoring Tool - The refactoring package *Package Refactoring* contains all of the refactoring supported by the Go Doctor, as well as types that are used to interact with those refactorings. The parameters for a specific refactoring must be mentioned in order to implement that refactoring on the code. For example in Rename refactoring, the user may select an identifier to rename, but the refactoring tool must also elicit, (1)

a new name for the identifier and (2) whether or not occurrences of the name should be replaced in the comments.

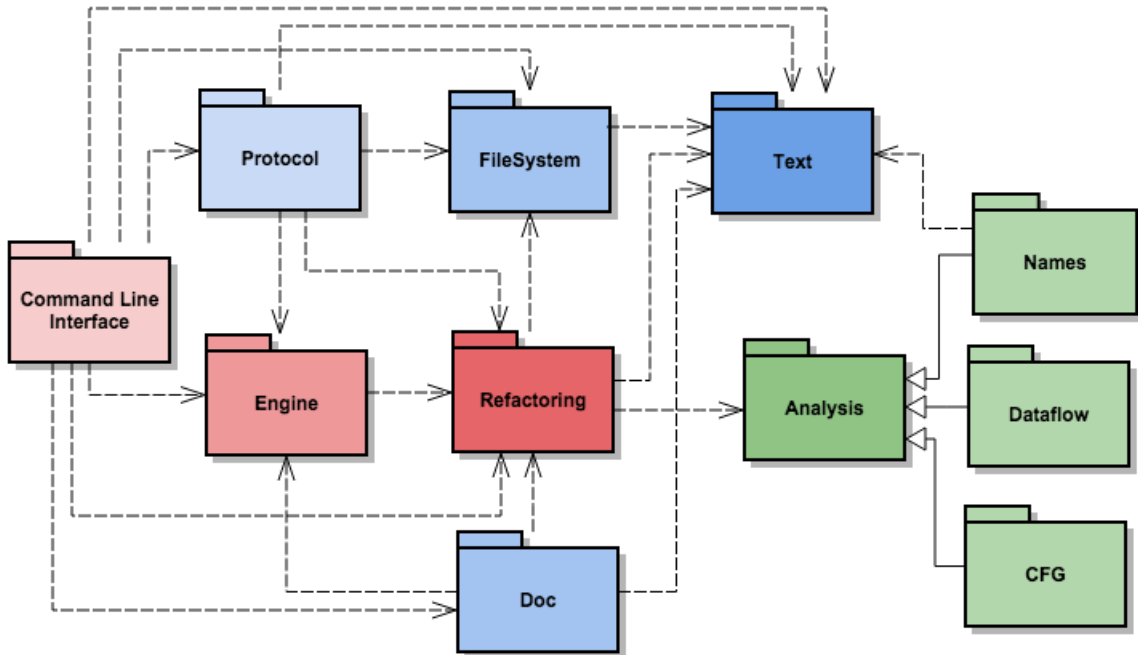


Figure 3.2: Package Diagram of Go Doctor

3.2 Extract Function Refactoring

The Extract Function refactoring extracts a sequence of statements from the existing block of code into a new function. There are two main parts to the Extract Function refactoring of the tool,

- Precondition Check
- Transformation

Before starting with the refactoring the user is expected to,

- Make sure the statement to be extracted is a part of the function that is, only statements inside a block(a function or if statement etc) can be extracted.

- Enter a valid function name for the new function the extracted code is put.

3.2.1 Precondition Check

1. **Valid Selection Check** - Whenever a set of statements are extracted, in order to check if the user made a valid selection the tool performs the following steps,
 - (a) Gets the Path Enclosing Intervals for the first statement of extracted code
 - (b) Get the Path Enclosing Intervals for the last statement of extracted code
 - (c) If the nodes in the path enclosing intervals match, then the selection is valid

```
package main

import "fmt"

func main() {
    a := 3
    b := 6
    c := a + b
    fmt.Println("C is ", c)
}
```

Figure 3.3: A Go Language Source Code

Path Enclosing Interval of a given statement refers to a list of nodes that encloses the given node or statement. These nodes that encloses the given node are all the ancestors of the node up to the AST root[61].

In Figure 3.3, the Path Enclosing Interval for the highlighted statement is ,

```
*ast.AssignStmt
*ast.BlockStmt
*ast.FuncDecl
*ast.File
```

Figure 3.4: Nodes of Path Enclosing Interval of highlighted statement in Figure 3.3

where **ast.AssignStmt* refers to the selected statement for which we are finding the Path Enclosing Interval for. Figure 3.5 gives the better representation of the the path enclosing interval of the selected statement “c:=a+b” which is an **ast.AssignStmt*. Since the statement extracted has the same set of enclosing intervals **ast.BlockStmt*, **ast.FuncDecl* and **ast.File*, this statement can be extracted.

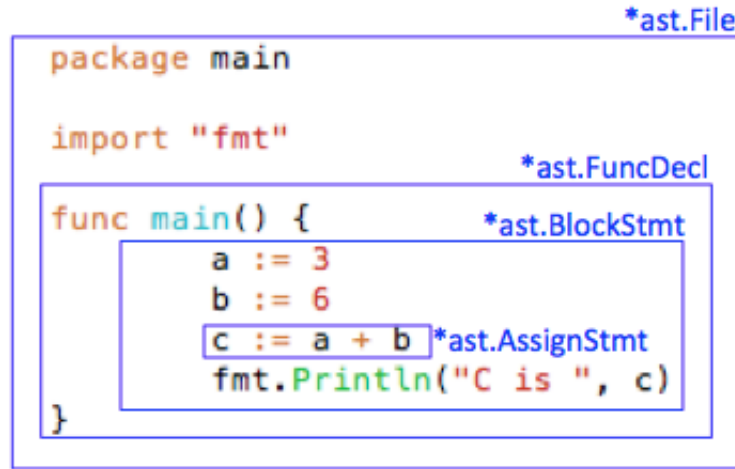


Figure 3.5: Path Enclosing Interval of highlighted statement in Figure 3.3

2. **Valid Statement Check** - The extracted code must not contain the following statements; if they do, the code fails.

- **Defer () Statements** - These statements are used to simplify functions and for clean up actions. This pushes the given function call to a list, and the given list of function calls are executed after the surrounding function returns[62]. Hence extracting a defer statement would cause the deferred function call to dispatch when the extracted function returns, which could change the behavior of the program. Figure 3.6 gives an example of how defer statements are used and Figure 3.7 describes the output of the program in Figure 3.6 based on the order the defer statements calls specific functions.

```

package main

import "fmt"

func fA() {
    fmt.Println("this is function A")
}

func fB() {
    fmt.Println("this is function B")
}

func main() {
    defer fA()
    defer fB()
}

```

Figure 3.6: An Example for Defer Statement

```

    this is function B
    this is function A

```

Figure 3.7: Output of Program in Figure 3.6

- **Return Statements** - When a *return* statement alone is extracted from the original function into a new function, the control would return from the extracted new function rather than the original function, and this may not preserve the behavior of the code[63].

```

func main() {
    x := 5
    fn := func() {
        fmt.Println("x is", x)
        x = x + 20
        fmt.Println("x is ", x)
    }
    fn()
    x++
    fn()
}

```

Figure 3.8: Example of Anonymous function

- **Anonymous functions** - Those functions that are useful when you want to define a function inline without having to name it. This mainly refers to creating a function inside of another function. The *Control Flow Graph(CFG)* that is built from the code, only uses a single function as input based on which it determines

the variables dataflow analysis, the Go Doctor is limited to normal functions and the tool still explores the possibilities of accommodating more than one function in its control flow graph. Figure 3.8 is the example of an anonymous function.

3. **Single Entry Single Exit Criteria** - In a control flow graph G , two nodes x and y are said to be enclosed in a single entry and single exit region if,

- x dominates y ,
- y post dominates x ,
- x and y are cycle equivalent in G .

The first two rules ensures that when control reaches y whenever it reaches x and vice verse. The third rules ensures that whenever control reaches x , it reaches y before reaching x again and vice verse. The author uses the ordered pair (x,y) to denote the *Single Entry Single Exit (SESE)* regions where x is the entry node and y is the exit node[64].

A piece of code is likely to be easily understood if it has only one entry point and one exit point at the bottom of its listing. A set of statements that are supposed to be extracted is said to have more than one entry or exit point if,

- (a) There are one or more *return* statements in the extracted code.
- (b) There are branch or jump statements like *break*, *goto* and *continue* in the extracted code.
- (c) The branch statements can be handles as long as some conditions are met.
- (d) Checking for valid entry criteria of each node,
 - i. If there is just one predecessor, then the statement node can be extracted.
 - ii. If there is more than one predecessor then,
 - Check if the predecessor is a part of the extracted code.

- Check if there is only one other node apart from the extracted nodes is the predecessor of the selected node.
- (e) Checking for valid Exit criteria,
- If there is one successor, then the statement can be extracted.
 - If there is more than one successor then,
 - Check if the Successor is a part of the extracted code.
 - Check if there is only one other node apart from the extracted nodes is the successor of the selected node.

4. **Labeled Branch Statement Check** - Whenever a branch statements like *goto*, *break*, *continue* and *fallthrough* are encountered, they may or may not have labels. Labels can have its own has a set of statements below it. Some of the conditions that have been set up to handle branch statements are,

- If any of the branch statement with a *label* is extracted , then the label it refers to must also be extracted along with it like the example in Figure 3.9.



Figure 3.9: Extract Function Refactoring on a labeled branch statement *break*

- Whenever there is a *break* statement, it can only be extracted if it is inside of a *for*, *switch* or *select* statement block.
- Whenever a *continue* statement is encountered it should be inside of a *for* statement block if it has to be extracted

- Whenever a *goto* statement is encountered, then it must have a *label* statement extracted along with it.

3.2.2 Live Variable Analysis

1. Building a Control Flow Graph [48]

A *Control Flow Graph*(CFG) is a directed graph in which each node represents a basic block and each edge represents the flow of control between basic blocks. To build a control flow graph, we first need to identify the basic blocks, which is nothing but a sequence of consecutive statements in which the flow of control must enter in the beginning of the first node of the block and leave at the end or at the last node of the block without halt or possibility of branching except at the end. Once we construct the basic blocks, add the edges that represent the control flow between these basic blocks. The Go Doctor uses the abstract syntax tree of the program to construct CFG, where it considers each statement to be a basic block that is, each individual statement that is a part of a block (*expressions, function calls, assignment statements, etc.*) or a block on itself (*if, for, switch, select, type switch blocks, etc.*).

The Go Doctor then creates an *entry* and *exit* nodes and traverses the list of statements in a depth first search and create an adjacency list implemented as a map of blocks. Adjacent blocks are stored as predecessors and successors separately for control flow information. The last node of the program has an edge to the *exit* node, also if there are statements in any other part of the program that makes the program exit, those statements have an edge to the *exit* node as well. The CFG is created for a separate functions and hence the anonymous functions become just one statement in the CFG, the statements within the anonymous functions are not considered. Figure 3.10 is the example of a program and its corresponding control flow graph in Figure 3.11

```

package main

import "fmt"

func main(){
    i := 0
    for i <= 5 {
        if i == 3 {
            break
        }
        fmt.Println(i)
        i++
    }
    fmt.Println("after loop")
}

```

Figure 3.10: Extract Function Refactoring on Branch Statement *break*

The highlighted nodes in the *Control Flow Graph* refers to the extracted code in Figure 3.10. This code satisfies the SESE criteria and the branching statement *break* is inside a *for* loop and hence it can be extracted without altering the behavior of code.

2. Dataflow Analysis[49]

Once the CFG is built, the dataflow analysis is performed to find the definitions and uses of variables. A variable is said to be defined when the variable is assigned a value. This can either be done by using a short assign operator(`i := 5`) or by assigning value to the variable when they are being declared (`var i int = 5`). In dataflow analysis, the Go Doctor keeps track of the variables that are being assigned, these are variables whose values get changed anywhere in the code other than where they are being declared or created. A use for variable occurs whenever the value of the variable is fetched or used in the code.

The execution of a program is nothing but a set of transformation of the program state which consists of the values of the variables in the program. The input state is associated with the program point before the statement and the output state is



Figure 3.11: Control Flow Graph for Program in Figure 3.10

associated with the program point after the statement. The execution path from point p_1 to point p_n to be sequence of points $p_1, p_2, p_3, p_4 \dots p_n$,

- p_i is the point in code immediately preceding a statement and p_{i+1} is the point immediately following that statement, or
- p_i is the end of some block and p_{i+1} is the beginning of the successor block

In dataflow analysis the Go Doctor associates with every program point a dataflow value, which is a set of variable definitions. The Go Doctor denotes the dataflow value before and after each statement S by $IN[S]$ and $OUT[S]$ respectively. The dataflow

problem is to find a solution to a set of constraints on the $IN[S]$ s and $OUT[S]$ s for statement S . There are two sets of constraints,

- Those based on the semantics of the statements (*Transfer Function*) the relationship between dataflow values before and after an assignment statement
- Those based on the *flow of control* (which is what we are more concerned about)

Consider a basic block and the constraints due to control flow between basic blocks (i.e statement B) can be written as $IN[B]$ and $OUT[B]$ and if the dataflow values are information about a set of constants that may be assigned to a variable and if statement P is a predecessor of statement B then

$$IN[B] = \bigcup OUT[P]$$

If statement S is a successor of statement B then,

$$OUT[B] = \bigcup IN[S]$$

3.2.3 Live Variable Analysis in Detail

Given program X with Control Flow Graph G . We say that a definition d reaches the point p in G if there is a path in G from point immediately following d to p , such that d is not “killed” along that path. A definition of a variable b is killed at some node n if there is a definition of b at the statement that corresponds to n . We can find the points where the definitions of a variable reaches by considering each variable definition in the program individually, and map it along all paths from the definition until either the definition is killed or or the *exit* node of the program is reached, which states that the program exited.

To understand the reaching definition for every statement in the program, we need to set up the two constraints for every single statement. Consider a definition,

$$d : a = b+5$$

This statement “generates” a definition of variable a and kills all the other definitions in the program that define variable a , while leaving the remaining incoming definitions unaffected.

In Live Variable analysis,

- gen_d set is the set of variables used in the expression or the statement
- $kill_d$ set is the set of all variables assigned, that is, they are not live above that point since their value will be overwritten.

Since definition reaches a program point as long as there exists at least one path along with the definition reaches, $OUT[X] \subseteq IN[Y]$ whenever there is a control flow edge from X to Y .

- IN set for a node is a set of definitions in the program that reach the point immediately before the node.

$$IN[Y] = \bigcup_{\text{All of } X \text{ that are predecessors of } Y} OUT[X]$$

- OUT set for a node is the set of definitions in the program that reach the point immediately following the node. The OUT set for a node Y consists of all definitions that either (1) are generated in Y or, (2) reach the entry to node Z but are not killed within Y

$$OUT[Y] = gen_Y \cup (IN[Y] - kill_Y)$$

In Live Variable analysis, consider a variable x and point p , we try to find whether the value of x at p could be used in any expression along some path in the control flow graph starting at p . If so, x is live at p ; otherwise, x is dead at p . Consider a basic block B , the

dataflow equations can directly be written in terms of $IN[B]$ and $OUT[B]$, which represents the set of variables live at the points immediately before and after block B .

- def_B - Set of variables defined (declared and assigned) in B prior to any use of that variable in B .
- use_B - Set of variables whose values may be used in B prior to any definition of the variables.

As a consequence of the definitions, any variables in use_B must be considered live at the entrance to block B , while definitions of variables in def_B are dead at the beginning of B . The equations relating def and use to the unknowns IN and OUT are defined as follows,

$$IN[EXIT] = \phi$$

and for all basic blocks B other than EXIT,

$$\begin{aligned} IN[B] &= use_B \cup (OUT[B] - def_B) \\ OUT[B] &= \bigcup_{S \text{ a successor of } B} IN[S] \end{aligned}$$

3.2.4 Implementation of the Dataflow analysis in Go Doctor

When trying to implement the dataflow analysis on the Go source code, one has to identify those variables that are being assigned and those variables that are being defined. Some of the conditions that each variable needs to satisfy if it has to be placed in the right category is given below,

Defined

1. Defined - This set refers to those variables that are defined in the extracted code such as, The variable i is defined as an integer with value 5 using a *var* keyword declaration in the first line and *short assignment operator*($:=$) in the second line.

```
var i int = 5
j := 5
```

Figure 3.12: Variables Defined

```
var k int = 5
var i int = 7
k = 22
```

Figure 3.13: Variables Defined and Assigned

2. Assigned - This set refers to those variables that are assigned to in the extracted code, In the first two lines, variable k and i is defined to 5 and 7 respectively. In the third line variable k is assigned to the value 22. Hence there is a difference between variables that are defined and variables that are assigned.

```
type Pt struct {
    x, y int
}

func main() {
    p := Pt{3, 4}
    fmt.Println("Old Pt", p)
    p.x = 5
    p.y = 6
    fmt.Println("New Pt", p)
    fmt.Println("Point P is:", p)
}
```

Figure 3.14: Extraction of Partially Assigned Variables

3. Partially Assigned - While dealing with structures with more than one variable defined in the *struct*, there are certain things that needs to be considered and this is illustrated in Figure 3.14

When trying to extract the part of the code inside the red box into a new function which means, Pt is partially assigned inside function *main* and partially assigned inside function *foo* (which would be the name of the new function into which the code is

```

type Pt struct {
    x, y int
}

func main() {
    p := Pt{3, 4}
    fmt.Println("Old Pt", p)
    p.x = 5
    p = foo(p)
    fmt.Println("Point P is:", p)
}

func foo(p Pt) Pt {
    p.y = 6
    fmt.Println("New Pt", p)
    return p
}

```

Figure 3.15: Illustration of Partially Assigned Variables after Extraction

extracted into). In this case, the Go Doctor would pass the entire structure *Pt* as a parameter to the new function as seen in Figure 3.15. Consider the Figure 3.16,

```

p := Pt{3, 4}

```

Figure 3.16: Structure Declaration by Value

the variable *p* is assigned to the value of the structure *Pt*, and *p* can be passed by value into new function

- When passed as a value, any changes made to any variables of the structure , the structure variable has to be returned to the original function and if that's the only variable that is being return by the new function, then there should not be a short assign(=) operator used when trying to call the function, as the line in Figure 3.15

```

p = foo(p)

```

Figure 3.17: Function Call for Structure Passed by Value

- When there are more than one variable returned by the function, then we use the short assign statement

Sometimes, the definition of the structure variable could be a address location,

```
p := &Pt{3, 4}
```

Figure 3.18: Structure Declaration by Reference

this means, p is assigned to reference the struct Pt , and p has to be passed by reference into the new function

- when passed as a reference, any changes made to any variables of the struct, the struct variable need not be returned into the original function, since the value has already been modified. Hence there is no need to return the modified variable to the original calling function

```
foo(p)
```

Figure 3.19: Function Call for Structure Passed by reference

- When there are more than one variable returned by the function, then we use the short assign statement for the new variables that are returned but we never return the structure, since its value has already been changed.

Used

1. In Figure 3.20, line 2 the variable i which is on right side of $=$ or $:=$ operator is said

```
var i int = 6  
k := i
```

Figure 3.20: Variables Used

to be used. where as k is defined in line 2.

2. Those variables in index expressions $x[i]$.
3. Those variables in a selector expression such as, $x.Time$.

Variables Passed, Returned, and Defined

To compute the variables that need to be passed in as a parameter, returned from the new function or defined inside the new function, the Go Doctor manipulates the following variables,

1. *The set of variables alive at the first line of the extracted code* - Based on the live-in and live-out sets computed by the live variables analysis, the Go Doctor gets the set of variables that are live at the entry to the first statement of the extracted code. These variables are referred to as *ALIVE_FIRST*.
2. *Variables initialized at the for statement* - While checking for the live-in variables in the first statement of the extracted code, variables that are defined as a part of the *if*, *switch*, *typeswitch*, *range*, *for* statements and those variables are temporarily declared and used inside the about block statements. The set of such variables is referred to as *INIT_VARIABLES*.
3. *The set of variables that are alive at the last line of the extracted code* - This is possible only if the variables are being used in the statements after the selected set of statements. These variables will be referred to as *ALIVE_LAST*.
 - Based on the sets of live-in and live-out variables, the tool gets the set of variables that are live at the exit from the last statement of the extracted code.
 - However, if it is a *for* loop, the tool uses the live-in set of the statement immediately after the end of the for loop.
4. *Variables that are Defined, Assigned and Partially Assigned* - Where variables that are defined or declared in the selected statements are part of the *DEFINED* and variables whose values are changed in the set of selected statements but are not declared becomes a part of the *ASSIGNED* set.

5. *Variables that are used (but not declared or assigned) in the selected statements* - This is the *USED* set.

Now, the following variables are passed as parameters to the extracted function,

$$PARAMETER_LIST := (ALIVE_FIRST - INIT_VARIABLES) \cap USED.$$

In order to manipulate those variables that are returned from the extracted function,

$$RETURN_LIST := (ALIVE_LAST - PARAMETER_LIST) \cap DEFINED.$$

Finally, the following variables need to be defined inside the extracted function,

$$TEMP_VARIABLE := (ASSIGNED - PARAMETER_LIST) \cup (USED - ALIVE_FIRST)$$

$$VAR_DEFINED := TEMP_VARIABLE - INIT_VARIABLES$$

Function Call to Extracted Function

When returning arguments from a new function, if there is any new variable that is defined inside of the extracted function that is used in the original function, then the *shortAssignFlag* is set to true, this defines if the function call to the new function must have a ‘:=’ operator or just a ‘=’ operator when trying to return variables into the calling or the original function. For example in Figure 3.21 and 3.22.

```
func main() {  
    a := 5  
    b := 4  
    fmt.Println(b)  
    fmt.Println(a, b)  
}
```

Figure 3.21: Example Program for Function Call Replacement

```
func main() {  
    a := 5  
    b := foo()  
    fmt.Println(a, b)  
}  
func foo() int {  
    b := 4  
    fmt.Println(b)  
    return b  
}
```

Figure 3.22: Output Program Function Call Replacement

Chapter 4

Evaluation

4.1 Purpose

The purpose of my research is to prototype Extract Function refactoring adapted to the unique features of the Go language using dataflow analysis to ensure that behavior of code is not changed when a contiguous block of code is being extracted from a block of code inside a particular function. The Extract Function refactoring is implemented for Go programming language, which combines the efficiency of a statically typed language with the ease of programming in a dynamically typed language. A control flow graph is built from the abstract syntax tree (AST) of the program, the control flow graph is intraprocedural (it is limited to single function of the program) mainly to the function that contains the selected block of statements that is to be extracted. Dataflow analysis was performed on the control flow graph, which lists the variables that are live or killed in every statement of the function based on how and where the variables are defined to be used in the program.

Unit testing was used during the initial development. A testing platform was created which pulled in close to one million lines of Go language code on which the refactoring was made to run. The errors were noted, and their cause was also analyzed and decisions were made accordingly.

4.2 Testing Framework

Refactoring engines automate the application of refactoring, where the programmer just needs to select the refactoring they need to apply on the particular code they select, the engine will automatically check the preconditions and apply the transformation across

the entire program if the preconditions are satisfied. The reason why programmers are encouraged to use refactoring engines is because its the safest way of transforming a program, since manual refactoring is erroneous and cumbersome. It is critical that refactoring engines are reliable since the bug found in the engine might cascade its way to the transformed program.

There have been many techniques that were proposed that automates testing of refactoring engines, which includes both generation of test inputs and checking the test outputs[50]. One of which is the one where a general framework ASTGen which allows the programmers to write imperative generators whose executions produce complex input programs for refactoring engine. This makes the developer focus on the creative aspects of testing rather than the mechanical production of test inputs [51]. ASTGen follows the bound-exhaustive approach[52][53] for exhaustively testing all inputs within the given bound. This approach covers all corner cases within the given bound. However such techniques have several deficiencies. First, they require substantial manual effort for writing test generators. Second, the generated test inputs may not represent real refactoring scenarios; the tests are basically corners cases the IDE nor the programmers care about or hardly use. Third, they do not provide any estimate of how reliable refactoring engines are for tasks on real software projects.

Some researches proposes a testing refactoring engines and evaluating properties such as stability, efficiency, real time reliability can be tested by systematically applying refactoring on a large number of places in real software projects and keep track of the real time failures and take the necessary actions [54]. Their approach consists of,

1. Collect a set of real time projects, apply refactoring and collect the scenarios where the refactoring engine throws exception or produces code that doesnt compile.
2. Once the scenarios are collected, analyse them to check if they are valid errors and if they are duplicated.

3. Inspect the failures and minimize them and identify the non-duplicate bugs and report them.

4.2.1 Go Patient

Go Patient [55] is a tool that is used to test the Go Doctor. The process that is being used in the Go Patient tool is listed below,

1. Creating a Test Code Workspace - In order to begin with the testing, the Go Patient creates a separate test code workspace, that has nothing to do with the refactoring tools code, so that the developer is free to modify, add and delete test inputs in the workspace without affecting the refactoring code. The tool cloned the top 100 Github repositories that use Go programming and their dependencies. The Go Patient determines those packages that are runnable on the system and keeps track of them for testing on the tool.
2. Overview of the testing process - Once the testing workspace is created with the code from Github, the Go Patient creates a test planner based on,
 - A template that tells the test planner what commands to execute to run a test (the test number, Go package being tested, and the selected region).
 - The selection the Go Patient wants to test. For example, identifiers in the case of rename refactoring or contiguous set of statements in the case of Extract Function refactoring.

4.3 Test Code

The data or input programs on which Extract Function refactoring was performed took into account quite a few boundary conditions. A total of 274 test cases were written to verify the durability, stability and the error catching ability of the refactoring tool. There were a

lot of corner cases that had to be taken into account, some of which were very particular to the Go language syntax.

The data used to test the code was based on open source code in Go language that was adopted from GitHub. The programs that did not compile were removed then 100 test cases were created each time, and the result was documented and analyzed. The programs that failed compilation were ignored and removed from the array of codes that were set for testing.

4.4 Testing Results

Figure 4.1 and Table 4.1 is the result evaluation from more than 200 test cases.

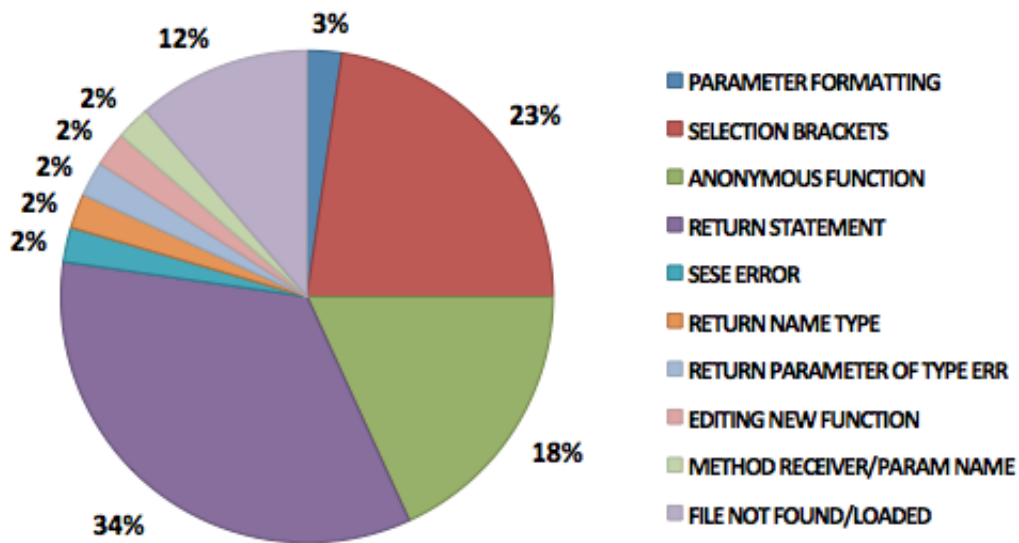


Figure 4.1: Graphical Report for Tests

Figure 4.1 provides a list of errors that were thrown during the testing phase and their percentage of occurrence. The reason for not handling *return* and *anonymous* statements are explained in following sections. The second highest set of errors was thrown since the

| S.No | Test Case Type | Fixed | Occurrence |
|------|-------------------------------------|-------|------------|
| 1 | PARAMETER FORMATTING | YES | 1 |
| 2 | SELECTION BRACKETS | YES | 10 |
| 4 | RETURN STATEMENT | NO | 15 |
| 5 | SESE ERROR | YES | 1 |
| 6 | RETURN NAME TYPE | YES | 1 |
| 7 | RETURN PARAMETER OF TYPE ERR | YES | 1 |
| 8 | EDITING THE NEW FUNCTION | YES | 1 |
| 9 | METHOD RECEIVER/PARAMETER SAME NAME | YES | 1 |
| 10 | FILE NOT FOUND/LOADED | NO | 5 |

Table 4.1: List of Test Cases

automated testing environment made a selection from the midpoint of a statement. This was rectified by selecting the entire statement under consideration. Other errors that occurred were elemental in nature that occurred after unit testing. Unit test cases were written for them, and these errors were fixed respectively.

4.5 Decision made based on Results

The Go Doctor does not handle *return statements*, *anonymous functions* and *defer statements*. These decisions were justified by analysing 4 million lines of Go Language code, where go patient[55] parsed through the statements in the go files to give a number of statements for *return*, *anonymous*, and *defer* is explained in Figure 4.2. The reason for not handling these statements are discussed below in detail.

4.5.1 Return Statements

It is evident that most number of errors thrown was due to the presence of the *return* statement in the extracted part of code. The *return* statement cannot be extracted since it tends to change the behavior of code [63]. Some of the places where *return* statement was commonly used as depicted in Figure 4.3. As seen in the bar graph, identifiers and function calls are the most commonly returned entities and when trying to extract this into a new function, the behavior of code changes, whenever the new function is called. The second

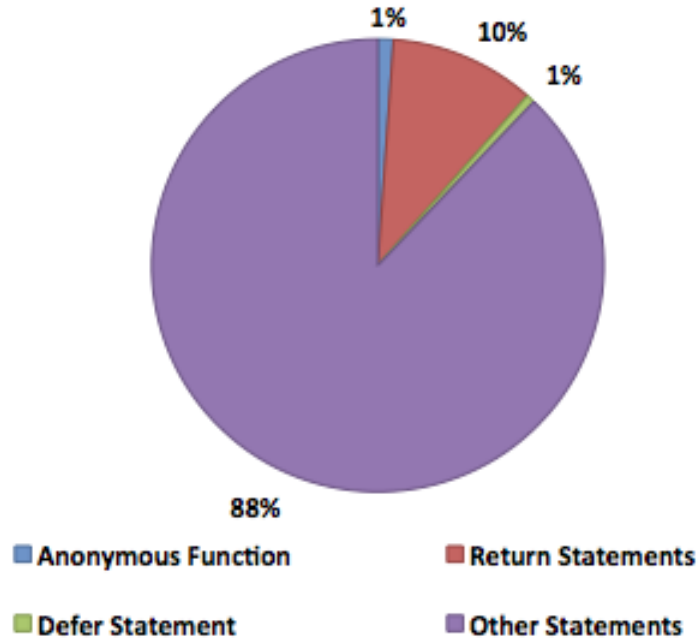


Figure 4.2: Proof for Decisions

most commonly returned entity is *none*, which means the return statement has no value to be returned. For example, in Figure 4.4, inside the *if* statement, if the variable *i* equals 6, then the program exits as the function returns from the *main* function, in the case Figure 4.5 after the *if* statement is extracted into the new function, when the *return* statement is executed, then control flow returns from the new function into the calling function which is the *main* function.

According to Max Schafer et al. [73], there are certain ways *return* statements can be handled,

1. When trying to extract block statements with *return* statements, such that the last statement extracted is the *return* statement. For example, Figure 4.6 the block statement inside the function *foo* can be extracted provided, the function call is returned from the calling function as on Figure 4.7.
2. Every statement in the extracted function must have the same predecessors and successors as before the extraction.

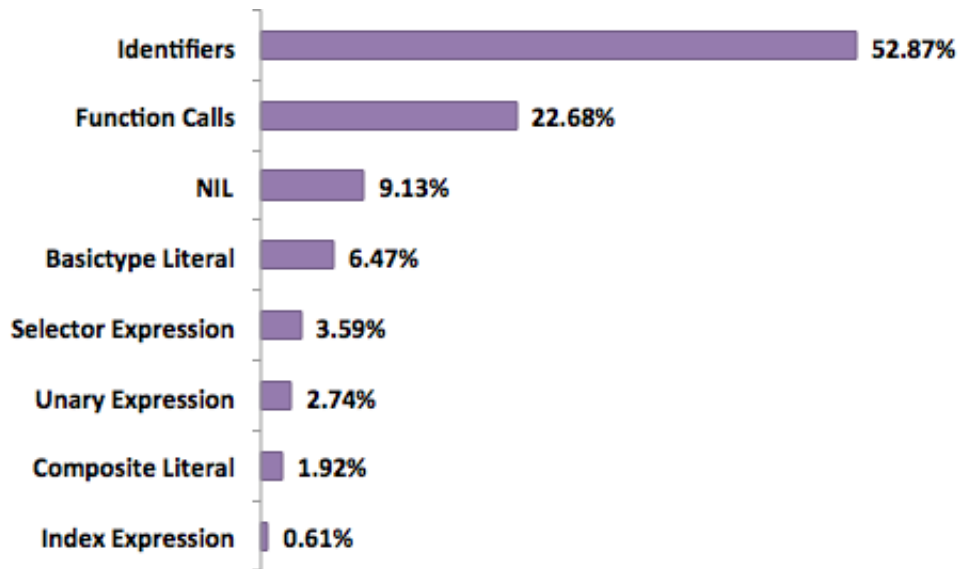


Figure 4.3: Return Statement Usage

```

func main() {
    i := 5
    i++
    if i == 6 {
        return
    }
    fmt.Println(i)
}

```

Figure 4.4: Program with Return Statement

4.5.2 Anonymous Functions

The control flow graph that is built from code, only uses a single function as input based on which it determines the variables dataflow analysis, hence the go doctor is limited to normal functions and the tool still explores the possibilities of accommodating more than one function in its control flow graph. Accordingly, an error is thrown when the extracted code has an anonymous function statement or is inside an anonymous function.

```

func main() {
    i := 5
    i++
    foo(i)
    fmt.Println(i)
}

func foo(i int) {
    if i == 6 {
        return
    }
}

```

Figure 4.5: Program with Extracted Return Statement

```

func main() {
    i := 5
    i += 2
    fmt.Println(foo(i))
}

func foo(i int) int {
    if i == 6 {
        return 10
    }
    return i + 1
}

```

Figure 4.6: Program with Return Statement that can be Extracted

4.5.3 Defer Statements

Extracting a *defer* statement would cause the deferred function call to dispatch when the extracted function returns, this may change the program behavior. The *Defer* statement moves function call to a list. This list of function calls that is saved is executed after the surrounding function returns. *Defer* is used to simplify functions that performs clean-up actions[72]. It is an effective way to deal with situations such as resources that must be released regardless of which path a function takes to return. Some of the situations where *defer* statement is commonly used is while unlocking a *mutex* or while closing a file[59]. Below is a thorough analysis of what each *defer* statements in the 4 million lines of code analyzed meant.


```

func main() {
    i := 5
    i += 2
    fmt.Println(foo(i))
}

func foo(i int) int {
    return n(i)
}

func n(i int) int {
    if i == 6 {
        return 10
    }
    return i + 1
}

```

Figure 4.7: Program with Extracted Return Statement from Figure 4.6

Where Selector Expression refers to an expression followed by a selector for example, *defer fileset.Close()*; While deferring identifier refers to deferring a function call for example, *defer teardown(c)*; Deferring caller expression with selector expression for example, *defer util.Run()()* and deferring caller expression with identifier meaning *defer teardown(c)()*; Deferring Inline function can have various implementations.

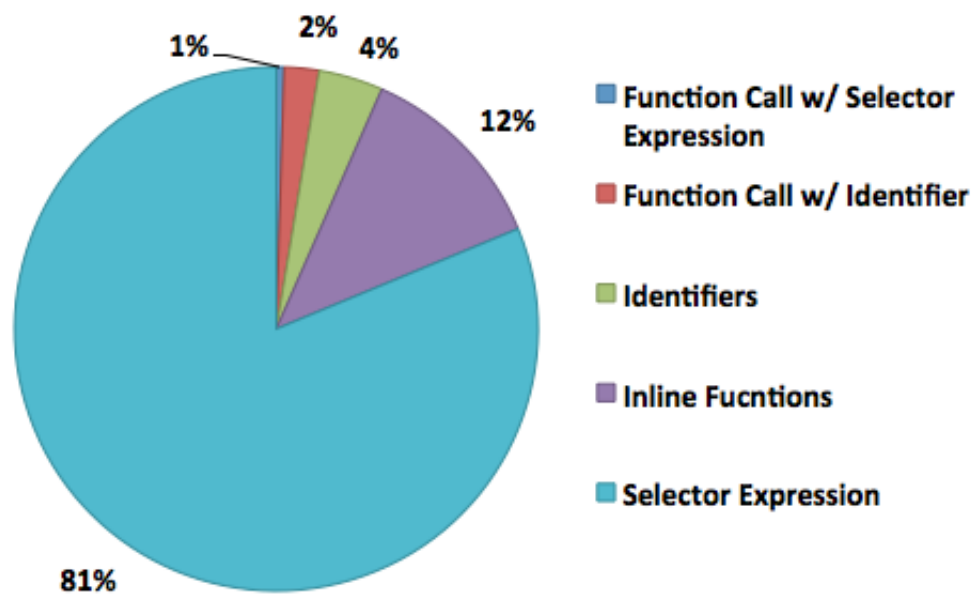


Figure 4.8: Defer Statement Usage

Chapter 5

Conclusion and Future Works

This thesis focuses on importance of code refactoring, a technique for restructuring the internal design of the code without changing its external behavior. One of the most commonly used refactoring is Extract Function refactoring. This explains the basics of function outlining in compiler which leads to the concept Extract Function refactoring.

Extract Function refactoring decomposes large code fragments into small cohesive functions. This is implemented by two main conditions checks, the precondition and variable analysis, where the precondition check, selection check, validation check, single entry single exit criteria, and variable analysis that determines the variables that needs to be passed, returned and declared inside the new function. This thesis lists out the limitation of the

```
func main() {  
    a := 5  
    b := 7  
    b = 10  
    fmt.Println(a, b)  
}
```

Figure 5.1: Removing Local Variable Declaration

Extract Function refactoring that was implemented for go language, where handling anonymous functions, *defer* statements and *return* statements for which proper explanation was given. In addition to that, this version of Go Doctor the user makes a selection and only that selection is extracted into a new function. The Go Doctor could parse through the entire program and find the statements that are just as same as the extracted statements and replace them with the function call to the new function.

Finally, one other functionality that could be added to the Go Doctor is removing the local variable declaration in a program. Consider Figure 5.1, here variable *b* is declared as

7, but even before it is used anywhere else in the program, it is reassigned to 10, hence the declared of variable b to 7 was invalid. Removing these invalid local variable declaration can help reduce memory usage.

Bibliography

- [1] Metrics and Models in Software Quality Engineering, 2nd Edition, Stephen H. Kan
- [2] Software Quality: Definitions and Strategic Issues, Ronan Fitzpatrick, 1996
- [3] M. E. Fagan. Design and code inspections to reduce errors in program development. IBM Systems Journal, 15(3):182-211, 1976.
- [4] Refactoring: Improving the Design of Existing Code by Martin Fowler, Kent Beck (Contributor), John Brant (Contributor), William Opdyke, Don Roberts
- [5] Java Quality Assurance by Detecting Code Smells by Eva van Emden, Leon Moonen
- [6] Integrating Code Smells Detection with Refactoring Tool Support by Kwankamol Nongpong - read this more
- [7] M. Fowler, Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [8] P. Weigerber and S. Diehl, Are Refactorings Less Error-Prone than Other Changes? Proc. Intl Workshop Mining Software Repositories, pp. 112-118, 2006.
- [9] K. Stroggylos and D. Spinellis, Refactoring-Does It Improve Software Quality? Proc. Fifth Intl Workshop Software Quality, pp. 10-16, 2007.
- [10] <http://www.versionone.com/agile-101/refactoring.asp>
- [11] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In Proceedings of the International Conference on Software Maintenance (ICSM 01), Florence, Italy, November 6-10, pages 736-743. IEEE Computer Society, Los Alamitos, California, November 2001.
- [12] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering, pages 1120. ACM Press, 2002.
- [13] Hayden Melton and Ewan Tempero. Identifying refactoring opportunities by identifying dependency cycles. In Proceedings of the 29th Australasian Computer Science Conference - Volume 48, ACSC 06. 2006.

- [14] P. Meananeatra, S. Rongviriyapanish, and T. Apiwattanapong. Using software metrics to select refactoring for long method bad smell. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2011 8th International Conference on*, pages 492–495. may 2011.
- [15] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *J. Softw. Maint. Evol.*, 20:435461, November 2008.
- [16] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 01), Lisbon, Portugal, March 1416*, pages 3038. IEEE Computer Society, Los Alamitos, California, March 2001.
- [17] Cyrille Artho and Armin Biere. Applying static analysis to large-scale multithreaded Java programs. In *Proceedings of the 13th Australian Software Engineering Conference (ASWEC 2001)*, pages 6875. 2001.
- [18] Stephen C. Johnson. Lint: a C program checker. In *Unix Programmings Manual*, pages 292303. 1978.
- [19] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 11*, pages 10371039. ACM, New York, NY, USA, 2011.
- [20] Schmager, F., Cameron, N., Noble, J.: GoHotDraw: evaluating the go programming language with design patterns. In: *Evaluation and Usability of Programming Languages and Tools, PLATEAU 2010*, pp. 10:110:6. ACM, New York (2010)
- [21] <http://www.hydrogen18.com/blog/golang-embedding.html>
- [22] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Reading, Massachusetts, USA, 1999
- [23] William F. Opdyke, *Refactoring object-oriented frameworks*, University of Illinois at Urbana-Champaign, Champaign, IL, 1992
- [24] Defining (reflexive) transitive closure on finite models, Jan van Eijck revised version: 9th of June, 2008
- [25] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag Berlin Heidelberg, New York, NY, USA, 1999.
- [26] Brian Foote and William F. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In *PLoP94: Proceedings of the 1st Conference on Pattern Languages of Programs*, pages 239257. Addison-Wesley, 1995.

- [27] Creating abstract superclasses by refactoring
- [28] Stas Negara , Nicholas Chen , Mohsen Vakilian , Ralph E. Johnson , Danny Dig, A comparative study of manual and automated refactorings, Proceedings of the 27th European conference on Object-Oriented Programming, July 01-05, 2013, Montpellier, France
- [29] Use, Disuse, and Misuse of Automated Refactorings
- [30] Program slicing Mark Weiser
- [31] Slicing Programs with Arbitrary Control Flow
- [32] Restructuring programs by tucking statements into functions
- [33] Identification of Extract Method Refactoring Opportunities (2009) by Nikolaos Tsan-talis , Alexander Chatzigeorgiou
- [34] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. TAPOS 97, Journal of Theory and Practice of Object Systems, 3(4):253263, 1997.
- [35] R. Komondoor, and S. Horwitz, "Effective, Automatic Procedure Extraction," 11th IEEE International
- [36] Mark Harman,David Binkley, Ranjit Singh, Robert M. Hierons, Amorphous procedure extraction , at Source Code Analysis and Manipulation, 2004. Fourth IEEE Interna-tional Workshop on Sept. 2004
- [37] Katsushisa Maruyama, Automated method-extraction refactoring by using block-based slicing. pages 31-40. ACM Press,2001.
- [38] Tom Mens, Serge Demeyer, Dirk Janssens, Formalising Behaviour Preserving Pro-gram Transformations. In Proc. First Intl Conf. Graph Transformation,pages 286-301. Springer-Verlag,2002
- [39] Mathieu Verbaere, Ran Ettinger and Oege de Moor JunGL: a Scripting Language for Refactoring. In : Proc. ICSE,pp. 172-181(2006)
- [40] Dmitry Jemerov ,Implementing Refactroings in IntelliJ IDEA
- [41] eclipse.org. Eclipse. <http://www.eclipse.org>, 2003
- [42] Ge, X., DuBose, Q.L., Murphy-Hill, E.: Reconciling manual and automatic refactoring. In: ICSE (2012)
- [43] <http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Fconcepts%2Fconcept-refactoring.htm>
- [44] <http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Fconcepts%2Fconcept-refactoring.htm>

- [45] sourcemaking.com/refactoring/extract-method
- [46] <http://talks.golang.org/2012/splash.article>
- [47] <http://golang.org/doc/faq/Origins>
- [48] Mary Jean Harrold, Gregg Rothermel, and Alex Orso. Representation and analysis of software. <http://www.ics.uci.edu/lopes/teaching/inf212W12/readings/rep-analysis-soft.pdf>
- [49] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman: Compilers Principles, Techniques, and Tools; 2nd ed., Addison-Wesley, 2007
- [50] Chandrasekhar Boyapati , Sarfraz Khurshid , Darko Marinov, Korat: automated testing based on Java predicates, Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, July 22-24, 2002, Roma, Italy. Pages 123-133
- [51] Brett Daniel , Danny Dig , Kely Garcia , Darko Marinov, Automated testing of refactoring engines, Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, September 03-07, 2007, Dubrovnik, Croatia
- [52] Darko Marinov, "Automatic Testing of Software with Structurally Complex Inputs", 2005
- [53] Kevin, S., Jinlin, Y., David, C. (2005) Software assurance by bounded exhaustive testing[J]. IEEE Transactions on Software Engineering 31: pp. 328-339
- [54] Milos Gligoric , Farnaz Behrang , Yilong Li , Jeffrey Overbey , Munawar Hafiz , Darko Marinov, Systematic testing of refactoring engines on real software projects, Proceedings of the 27th European conference on Object-Oriented Programming, July 01-05, 2013, Montpellier, France
- [55] <https://github.com/joverbey/gopatient>
- [56] Kenneth Rosen, "Discrete Mathematics and Its Applications"; 7th edition, McGraw-Hill, 2011
- [57] <https://github.com/godoctor/godoctor/tree/master/analysis/names>
- [58] <http://www.cs.odu.edu/toida/nerzic/content/relation/closure/closure.html>
- [59] https://golang.org/doc/effective_go.html
- [60] <https://github.com/godoctor/godoctor>
- [61] <https://godoc.org/golang.org/x/tools/go/ast/astutil>
- [62] <http://blog.golang.org/defer-panic-and-recover>
- [63] J. L. Overbey, "A toolkit for constructing refactoring engines," Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 2011

- [64] Single Entry Single Exit and Control Regions in Linear Time, by Richard Johnson, David Pearson, Keshav Pingali, Cornell University
- [65] <http://talks.golang.org/2012/splash.article>
- [66] <http://golang.org/doc/faq>
- [67] T. Mens and T. Tourwe. A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2): 126-162, February 2004.
- [68] Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall PTR, Upper Saddle River, NJ, 2008
- [69] K. Pettis and R. C. Hansen. Prole guided code positioning. In Programming Language Design and Implementation (PLDI), pages 1627, 1990.
- [70] T. Sukanuma, T. Yasue, and T. Nakatani. A region based compilation technique for a java just-in-time compiler. In Programming Language Design and Implementation (PLDI), pages 312323, 2003.
- [71] P.Zhao and J.N.Amaral, Ablego: a function outlining and partial inlining framework: Research articles, Softw. Pract. Exper., vol. 37, no. 5, pp. 465491.
- [72] <http://blog.golang.org/deferpanicandrecover>
- [73] Max *Schäfer* , Mathieu Verbaere , Torbjørn Ekman , Oege Moor, Stepping Stones over the Refactoring Rubicon, Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, July 06-10, 2009, Italy

Appendix A

Toggle Variable Refactoring

A ToggleVar refactoring converts between explicitly-typed variable declarations (`var n int = 5`) and short assignment statements (`n := 5`). The user makes a valid statement selection that he wants convert and this statement should either be a assignment statement, that have to be converted into a variable declaration statement or vice-verse.

A.1 Precondition Check

The selected statement must be an assignment statement or a variable declaration. Assignment statements that are a part of another statements such as conditional statements etc cannot be used.

A.2 Working

When an assignment statement with a short assignment symbol is selected,

- The name of the variables in the Left Hand Side of the assignment statement are saved to an array
- The type of the variables, or expression on the right hand side of the assignment statement is calculated and stored into an array
- The replacement strings are created based on the Left Hand Side of the assignment statement array and right hand side of the assignment statement array that is returned.
- Whenever there is a function on the right hand side of the assignment statement that returned two values of different type, when toggled, the type of the variables are not mentioned.
- Whenever there is a function on the right hand side of the assignment statement that returned two values of same type, when toggled, the type of the variables are mentioned.
- Whenever there are two different expressions on right hand side of the assignment statement that return two different types, they are split into two assignment statements

```

func f() (string, float64) {
    return "hello", 2.3
}

func main() {
    i, x := f()
    fmt.Println(i, x)
}

```

➔

```

func f() (string, float64) {
    return 1, 2.3
}

func main() {
    var i, x float64 = f()
    fmt.Println(i, x)
}

```

Figure A.1: Multivalued Function

```

func f() (float64, float64) {
    return 1, 2.3
}

func main() {
    i, x := f()
    fmt.Println(i, x)
}

```

➔

```

func f() (float64, float64) {
    return 1, 2.3
}

func main() {
    var i, x float64 = f()
    fmt.Println(i, x)
}

```

Figure A.2: Multivalued Function returns of the Same Type

```

func main() {
    i, j := 3.5+6, 7+1
    fmt.Println(i, j)
}

```

➔

```

func main() {
    var i float64 = 3.5 + 6
    var j int = 7 + 1
    fmt.Println( i, j)
}

```

Figure A.3: Multiple Expression Declaration